TEKNILLINEN KORKEAKOULU

Sähkö- ja tietoliikennetekniikan osasto

Timo Karilinna

Analysointityökalun käytettävyyden parantaminen

Diplomityö, joka on jätetty tarkastettavaksi diplomi-insinöörin tutkintoa varten

Espoossa 18. tammikuuta 2002

Työn valvoja ja ohjaaja: Professori Nisse Husberg

TEKNILLINEN KORKEAKOULU

DIPLOMITYÖN TIIVISTELMÄ

Tekijä: Timo Karilinna

Työn nimi: Analysointityökalun käytettävyyden parantaminen

Englanninkielinen nimi: Improving the usability of an analysis tool

Päivämäärä: 18. tammikuuta 2002	**Sivumäärä:** 66
Osasto: Sähkö- ja tietoliikennetekniikka	**Professuuri:** Tik-79 Tietojenkäsittelyteoria
Valvoja: Prof. Nisse Husberg	**Ohjaaja:** Prof. Nisse Husberg

Tiivistelmä:

Rinnakkaisten ja hajautettujen järjestelmien verifiointiin on yli kahden vuosikymmenen tutkimuksen tuloksena saatu aikaan useita formaaleja menetelmiä, joista tässä työssä käsitellään Petri-verkkoihin ja niistä muodostettuihin saavutettavuusgraafeihin perustuvaa analyysiä.

Olemassa olevat analysointimenetelmät edellyttävät kuitenkin käyttäjältään yleensä taustalla olevien formalismien tuntemusta, eivätkä siten ole tulleet kovin laajan insinöörikunnan käyttöön. Tähän on vaikuttanut lisäksi se, että analyysityökalujen ohjelmoijat ovat keskittyneet varsinaisten formalismien ja algoritmien mahdollisimman tarkkaan ja tehokkaaseen toteutukseen ja käyttöliittymien saattaminen helposti omaksuttavaksi on jäänyt toissijaiseksi päämääräksi.

Tässä työssä tarkastellaan käyttöliittymien suunnitteluperiaatteita ja esitetään laadittuun graafiseen käyttöliittymään pohjautuva malli siitä, miten saavutettavuusanalyysiin perehtymätön henkilö voi helposti käyttää erästä aikaisemmin kehitettyä ja jo käytössä ollutta analysointityökaluohjelmaa. Lisäksi työhön sisältyen on muutettu eräs vanhempi työkaluohjelma tuottamaan TNSDL-mallista kyseisen analysointityökalun hyväksymiä Petri-verkkomallitiedostoja.

Avainsanat:

EMMA, käyttöliittymä, MARIA, Petri-verkko, PROD, saavutettavuusanalyysi, SDL, TNSDL

HELSINKI UNIVERSITY OF TECHNOLOGY

ABSTRACT OF THE MASTER'S THESIS

Author: Timo Karilinna	
Name of the Thesis: Improving the usability of an analysis tool	
Name in Finnish: Analysointityökalun käytettävyyden parantaminen	
Date: 18 January 2002	**Pages:** 66
Department: Electrical and Communications Engineering	**Professorship:** Tik-79 Theorethical Computer Science
Supervisor: Prof. Nisse Husberg	**Instructor:** Prof. Nisse Husberg

Abstract:

More than two decades of research and development have produced several formal methods for verification of concurrent and distributed systems. This thesis is mainly about reachability analysis of Petri nets.

Existing methods require, however, that the users are more or less familiar with the underlying formalisms and thus have not reached a widespread use among engineers. Another reason for this has been that the programmers of analysis tools have concentrated on implementing the formalisms and algorithms as accurately and effectively as possible, and making a user-friendly interface has become a secondary target.

This thesis studies general design principles of computer user interfaces and introduces a prototype graphical user interface to help a person, unfamiliar with reachability analysis, more easily use a previously developed and widely used analysis tool. In addition, an old Petri net translator has been modified to produce model files for the analyser from TNSDL files.

Keywords:

EMMA, human-computer interface, MARIA, Petri net, PROD, reachability analysis, SDL, TNSDL

Esipuhe

Tämä diplomityö on tehty TKK:n tietojenkäsittelyteorian laboratoriossa keväällä 2001 toimiessani siellä osapäiväisenä tutkimusapulaisena prof. Nisse Husbergin alaisuudessa. Esitänkin hänelle erityiset kiitokset, ensinnäkin tilaisuudesta päästä työskentelemään tämän mielenkiintoisen tutkimusalueen parissa, toiseksi hänen uskostaan ja jatkuvasta kannustuksestaan työtäni kohtaan.

Työ on tehty osana MARIA-projektia, jonka rahoittajina ovat olleet TEKES, Nokian Tutkimuskeskus, Nokia Telecommunications, Elisa Communications ja Ratahallintokeskus.

Useista viime vuosina tehdyistä laboratorion diplomitöistä poiketen tämä työ ei ole erityisen teoreettinen, vaan pyrkii tuomaan esiin ja osaltaan myös ratkaisemaan käytännön insinöörityössä ilmeneviä ongelmia formaaleja menetelmiä käytettäessä. Toivon, että työ herättää lukijoissaan ajatuksia ja synnyttää ideoita, jotka johtavat tulevina vuosina yhä parempiin ja helppokäyttöisempiin analysointityökaluihin.

Haluan kiittää kaikkia laboratorion tutkijoita ja muuta henkilökuntaa heiltä saamistani lukuisista neuvoista ja kommenteista, joita ilman työni ei olisi onnistunut. Erityisesti heistä haluan mainita DI Teemu Tynjälän, jonka kanssa olen usein käynyt pitkiä keskusteluja tekemäni kokeellisen käyttöliittymän ominaisuuksista ja sen käyttökelpoisuudesta mm. hänen työnsä tukena. Arvostan myös TkL Marko Mäkelältä saamaani apua ohjelmointiteknisissä ja MARIA:a koskevissa ongelmissa.

Lopuksi haluan kiittää puolisoani Saaraa, joka on jaksanut kestää sitä, että ajatukseni ovat usein harhailleet opintojeni ja tämän työn parissa, vaikka kotonakin olisi ollut paljon tekemistä.

Timo Karilinna
Helsingin Pakilassa, 18. tammikuuta 2002

Sisällysluettelo

Kuvaluettelo

Käytetyt merkinnät ja lyhenteet

$\Leftrightarrow$	Looginen ekvivalenssi
$\Rightarrow$	Looginen implikaatio
$\neg$	Looginen negaatio
$\wedge$	Looginen konjunktio
$\vee$	Looginen disjunktio
$\cup$	Joukkojen yhdiste (unioni)
$\cap$	Joukkojen leikkaus
$\times$	Joukkojen karteesinen tulo
$\varnothing$	Tyhjä joukko
$\mathbf{N}$	Luonnollisten lukujen joukko (1, 2, 3, ...)
$\exists$	Eksistenssikvanttori ("on olemassa")
$\forall$	Universaalikvanttori ("kaikilla")
$\models$	Toteutusrelaatio ("toteuttaa")
$\not\models$	Toteuttamattomuusrelaatio ("ei toteuta")
$\bigcirc$	Modaalioperaattori *Seuraava* (Next Time)
$\Diamond$	Modaalioperaattori *Lopulta* (Eventually)
$\Box$	Modaalioperaattori *Aina tästä alkaen* (Henceforth)
$\mathbf{U}$	Modaalioperaattori *Kunnes* (Until)
$\mathbf{V}$	Modaalioperaattori *Vapauttaa* (Release)

API	application programming interface
AWT	abstract window toolkit
CCITT	Comité Consultatif International Telégraphique et Teléphonique
EMMA	extendible multi method analyzer
FCF	flow control format
HCI	human-computer interaction
HTML	hypertext markup language
ISO	International Organization for Standardization
ITU	International Telecommunication Union
LBT	LTL to Büchi automaton translator
LTL	lineaarinen temporaalilogiikka (myös: linear temporal logic)
MARIA	modular reachability analyzer
P/T	paikka/transitio (myös: place/transition)
pid	process identifier
Pr/T	predikaatti/transitio (myös: predicate/transition)
SDL	specification and description language
SSH	secure shell
TESTP	TNSDL external symbol table presentation
TNSDL	Telenokia SDL

1. Johdanto

Nykyaikaisissa laajoissa tietojärjestelmissä, kuten tietoliikennesovelluksissa, jotka ovat rinnakkaisia ja hajautettuja, on suurena ongelmana verifioida eli näyttää oikeaksi ohjelmistojen toiminta. Menetelmät, jotka ovat hyvinkin käyttökelpoisia perinteisten peräkkäisrakenteisten ohjelmien testauksessa ja useimmiten paljastavat suurimman osan niiden virheistä, ovat riittämättömiä käytettäväksi silloin, kun ohjelmistot sisältävät synkronoimatonta rinnakkaisuutta.

Syynä ongelmiin on ensinnäkin, että rinnakkaiset ja hajautetut järjestelmät voivat saavuttaa niin suuren määrän erilaisia tiloja, että niiden hallitseminen on ihmisvoimin mahdotonta, mutta erityisesti se, että tiettyjen tilasekvenssien esiintyminen voi olla niin harvinaista, että niiden toistumista ei saada testaustilanteessa aikaiseksi. Tarvitaan formaaleja verifiointimenetelmiä, joilla automaattisesti luodaan järjestelmän tila-avaruus, tai ainakin tarvittava osa siitä, sekä erityisiä työkaluohjelmia haluttujen toimintojen ja ominaisuuksien tarkistamiseksi tila-avaruutta tutkimalla. Menetelmästä käytetään yleisesti nimitystä *saavutettavuusanalyysi*, joka tarkoittaa sitä, että tietokoneen avulla ratkaistaan järjestelmän kaikki saavutettavissa olevat tilat tai suotuisissa tapauksissa vain oleellinen osa niistä ja etsitään tiettyjä virhe- tms. erikoistiloja, kuten esimerkiksi lukkiutumia, ja kyseisiin tiloihin johtavia polkuja.

Teknillisen korkeakoulun tietojenkäsittelyteorian (aiemmin digitaalitekniikan) laboratoriossa on pitkään tutkittu ja kehitetty formaaleja menetelmiä rinnakkaisten ja hajautettujen järjestelmien analysointiin. Erääksi hyviä tuloksia tuottavaksi menetelmäksi on osoittautunut ja pääasialliseen käyttöön vakiintunut *Petri-verkkojen* ja niistä muodostettujen *saavutettavuusgraafien* analysointi tarkoitukseen kehitettyjen työkaluohjelmien avulla.

Petri-verkkokuvauksen laatiminen järjestelmästä edellyttää huomattavaa asiantuntemusta ja vasta pitkän kokemuksen myötä se onnistuu virheettömästi. Lisäksi kuvauksen laajuus kasvaa järjestelmän myötä ja riittävän laajasta järjestelmästä kuvausta ei enää kannata tehdä käsin. Siksi on myös kehitetty työkaluohjelmia Petri-verkkokuvauksen muodostamiseksi tietokoneen avulla lähtien jostain sovellusalueen asiantuntijoille tutusta kuvauskielestä. Hyviä tuloksia on saatu *SDL-kielen* ja sen johdannaisen *TNSDL:n* kääntämisestä Petri-verkkokuvauksiksi.

1.1. Tutkimusongelma

Analysointityökaluohjelmien käyttö ei ole, huolimatta niiden laajalti tunnustetusta erinomaisuudesta, yleistynyt toivotulla tavalla. Erääksi merkittäväksi syyksi on arveltu niiden vaikeasti omaksuttavia käyttöliittymiä, jotka ovat tähän asti olleet komentorivipohjaisia. Tämän päivän tietokoneen käyttäjät ovat jo niin tottuneita ikkunoituihin, hiirellä ohjattuihin valikoihin ja graafisiin elementteihin

perustuviin käyttöliittymiin, että kaikkinainen näppäimistön käyttö koetaan helposti vastenmieliseksi.

Formaaleja menetelmiä käytettäessä ongelmana on lisäksi se, että sovellusalueen asiantuntijat eivät yleensä omaa riittävää teoreettista tietämystä omaksuakseen tarvittavien analysointityökalujen käytön. Toisaalta heiltä ei sellaista voi odottaakaan, sillä saavutettavuusanalyysin pohjana olevat formalismit ja algoritmit muodostavat erittäin laajan ja teoreettisen tutkimusalan, jonka perusteidenkin hallinta edellyttäisi käytännön insinööreiltä liiallista lisäkoulutusta. Parempi vaihtoehto olisi kätkeä formalismi mahdollisimman hyvin työkalun sisään ja näyttää sekä tutkittava järjestelmä että analyysin tulokset käyttäjälle hänen omalla kielellään.

Käytännön järjestelmästä, jo varsin pienestäkin, tulee helposti erittäin suuri ja vaikeasti hallittava Petri-verkkomalli ja siitä edelleen jopa räjähdysmäisesti kasvava saavutettavuusgraafi. Olemassa olevat analyysityökalut sisältävät riittävästi toimintoja näiden tarkastelemiseksi, mutta komentorivipohjaiset käyttöliittymät tekevät työskentelyn hitaaksi ja tulosten esittäminen on erittäin rajoittunutta ja siten useimmiten liian epähavainnollista.

1.2. Työn tavoite

Tässä diplomityössä esitetään eräs näkemys siitä, miten Petri-verkkoihin perustuvien analysointityökalujen käytettävyyttä voidaan parantaa ja siten saada niiden käyttäjäkuntaa laajennetuksi. Työtä määriteltäessä tavoitteiksi asetettiin seuraavien kahden, toisistaan poikkeavan käyttäjäryhmän saattaminen saavutettavuusanalyysin piiriin:

- Tietoliikenne- tai muut suunnitteluinsinöörit, jotka kuvaavat järjestelmiä omilla kuvauskielillään (esim. SDL), eivätkä varsinaisesti ole työnsä puolesta kiinnostuneita analyysin perustana olevista formalismeista.

- Tietojenkäsittelyteorian opiskelijat, jotka ovat vasta perehtymässä käytettyihin formalismeihin ja joutuvat tekemään harjoitustöitä, joissa analysoidaan rinnakkaisia ja hajautettuja järjestelmiä.

Ensin mainittua ryhmää palvelemaan tarvittiin käännöstyökalu, joka tuottaa halutusta kuvauskielestä, tässä tapauksessa TNSDL:stä analysaattoria varten sopivan mallin ja mikäli mahdollista, myös analyysin tulokset olisi pyrittävä esittämään käyttäjien ymmärtämässä muodossa.

Opiskelijoita varten analysaattorin käyttöliittymän tuli olla sellainen, että se esittäisi analyysiprosessia mahdollisimman havainnollisesti ja siten kynnys aloittaa analyysityökalun käyttö madaltuisi. Kunnianhimoisempana tavoitteena jatkokehitystä ajatellen voitaisiin edelleen pitää eräänlaisen tietokoneavusteisen itseopiskelupaketin luomista saavutettavuusanalyysiin perehtymiseksi.

Kaikkia käyttäjiä ajatellen päädyttiin vaatimukseen, että käyttöliittymän on vastattava nykyaikaisia henkilökohtaisissa tietokoneissa käytettäviä sovelluksia, ts. sen on oltava ikkunoitu, valikkopohjainen ja hiiren avulla käytettävä, eli ns. *graafinen käyttöliittymä*. Nimitystä käytetään nykyään yleisesti tällaisista käyttöliittymistä riippumatta siitä, esitetäänkö varsinaista tietoa graafisesti vai ei, kunhan käyttö perustuu näyttöelementtien osoittamiseen ja toimenpiteiden aktivointiin sitä kautta.

Erittäin tärkeänä vaatimuksena nähtiin *virhesietoisuus*. Tottumattoman käyttäjän on todettu tekevän lukuisia mallinnusvirheitä, joista ohjelman on kyettävä selviytymään ja samalla pyrittävä opastamaan käyttäjää mahdollisimman havainnollisesti.

1.3. Työn sisältö

Luvussa 2 esitellään ensin lyhyesti ja epäformaalisti Petri-verkot, saavutettavuusgraafi, lineaarinen temporaalilogiikka LTL sekä järjestelmien kuvauskieli SDL ja sen johdannainen TNSDL. Luvussa 3 luodaan katsaus analysointityökalujen kehitykseen ja nykytilanteeseen ja esitellään tarkemmin tässä työssä käsitellyt työkaluohjelmat PROD, EMMA ja MARIA. Luvussa 4 tarkastellaan kirjallisuustutkimuksen pohjalta käyttöliittymien yleisiä suunnitteluperiaatteita ja käytettävyyden parantamista lähtien inhimillisistä tekijöistä ja päätyen nykypäivän graafisiin käyttöliittymiin.

Työssä käsitellyn tutkimusongelman varsinaisen ratkaisun muodostaa tietojenkäsittelyteorian laboratorion uusimpaan saavutettavuusanalysaattoriin MARIA:an tehty kokeellinen graafinen käyttöliittymä XMARIA[1]. Lisäksi laboratorion eräs vanhempi työkaluohjelma EMMA on työhön sisältyen muutettu tuottamaan tietoliikenneinsinöörien käyttämästä TNSDL-kielestä Petri-verkkokuvauksia MARIA:lle aikaisemman PROD:in sijasta. Tämä osuus työstä tähtää siihen, että jatkossa tutkittava järjestelmä voitaisiin syöttää analysoitavaksi TNSDL-kielellä ja käyttöliittymän avulla piilottaa Petri-verkkoesitys mahdollisimman tarkoin. Näiden tehtävien ratkaisuperiaatteita kuvataan luvuissa 5 ja 6 ja XMARIA:n ominaisuuksia ja käyttöä luvussa 7.

Luvussa 8 tarkastellaan saavutettuja tuloksia ja luodaan katsaus jatkokehitysnäkymiin.

1.4. Rajoituksia

Jotta käsiteltävä ongelmakenttä ei olisi paisunut liian laajaksi, päädyttiin analyysin pohjana olevien Petri-verkkojen ja saavutettavuusgraafien edes osittainenkin graafinen esittäminen jättämään jatkokehityksen aiheeksi. Toisaalta tähän tarkoitukseen on jo olemassa useita muita työkaluja ja työtä tehtäessä itse

[1] X-alkuisia ohjelmanimiä käytetään yleisesti ilmaisemaan graafista käyttöliittymää.

MARIA:ankin alkoi samanaikaisesti kehittyä visualisointiominaisuuksia, jotka hyödyntävät suoraan ohjelman sisäisiä tietorakenteita ja omaavat siten paremmat edellytykset tuottaa tehokkaasti ja nopeasti graafisia esityksiä kuin nyt tehty käyttöliittymä.

TNSDL-Petri-verkkokääntäjä EMMA:n muutostyössä rajoituttiin toteuttamaan ainoastaan riittävä määrä piirteitä sen osoittamiseksi, että uudella analysaattorilla tehty saavutettavuusgraafi vastaa vanhalla analysaattorilla samasta lähdetiedostosta tehtyä ja että graafisella käyttöliittymällä voidaan myös näiden käytäntöä lähellä olevien mallien tarkastelua helpottaa.

2. Teoreettista taustaa

Tässä luvussa esitellään suppeasti ne teoreettiset taustatiedot, joihin tässä työssä käsitellyt rinnakkaisten ja hajautettujen järjestelmien analyysimenetelmät perustuvat. Seuraavassa luvussa esitellään näitä menetelmiä käyttävät työkalut.

2.1. Petri-verkot

Petri-verkko on formalismi, joka on suunniteltu mallintamaan toisiinsa vaikuttavia rinnakkaisia komponentteja sisältäviä järjestelmiä. Alkuperäisen ajatuksen esitti Carl A. Petri vuonna 1962 väitöskirjassaan ja sen jälkeen lukuisat tutkijat ovat edelleen kehittäneet Petri-verkkojen teoriaa ja ne ovat vakiintuneet laajaan käyttöön. Petri-verkkojen eräs etu järjestelmiä kuvattaessa on se, että niiden osia tai suppeita verkkoja jopa kokonaisuudessaankin voidaan havainnollistaa graafisesti. Järjestelmien koon kasvaessa tämä etu tosin menettää hyvin pian merkitystään.

Petri-verkot luokitellaan karkeasti kahteen ryhmään, *matalan* ja *korkean tason* verkot. Matalan tason verkoista puhuttaessa tarkoitetaan esimerkiksi *paikka/transitio-* eli *P/T-verkkoja*. Korkean tason verkkoja on lukuisia eri lajeja, joista tässä työssä käsitellään *predikaatti/transitio-* eli *Pr/T-verkkoja* sekä *algebrallisia järjestelmäverkkoja*. Eräs kirjallisuudessa laajalti esiintyvä korkean tason verkkojen laji on *värilliset* tai *väritetyt Petri-verkot*[1].

Tässä esitetty suppea, mahdollisimman epäformaali esitys pohjautuu pääasiassa lähteisiin [Jen97] ja [LHK01]. Tarkoituksena on antaa lukijalle riittävät pohjatiedot Petri-verkoista ja niiden käytöstä järjestelmien analysoinnissa.

2.1.1. Paikka/transitio-verkot

Petri-verkon $N = (S, T; F)$ kolme pääkomponenttia ovat

- *Paikka* $s \in S$, jota kuvataan yleensä graafisesti ympyrällä tai soikiolla.
- *Transitio* $t \in T$, jota kuvataan poikkiviivalla tai suorakulmiolla.
- *Kaari* $e \in F$, jota kuvataan nuolella. $F \subseteq (S \times T) \cup (T \times S)$ on *vuorelaatio*.

S on paikkojen ja T transitioiden joukko, joille pätee: $S \cap T = \varnothing$. Paikkoja ja transitioita voi verkossa olla mielivaltainen määrä ja kaaret yhdistävät niitä vuorelaation mukaisesti siten, että jokaisen kaaren päissä on oltava erilaiset komponentit, ts. kaari johtaa aina joko paikasta transitioon tai päinvastoin. Lisäksi esitetään usein vaatimus, että jokaiseen transitioon on tultava vähintään yksi *esikaari* ja siitä on lähdettävä vähintään yksi *jälkikaari*. Paikkaa, josta johtaa kaari

[1] engl. coloured Petri nets

tiettyyn transitioon kutsutaan ko. transition *esipaikaksi* ja vastavasti määritellään käsite *jälkipaikka*.

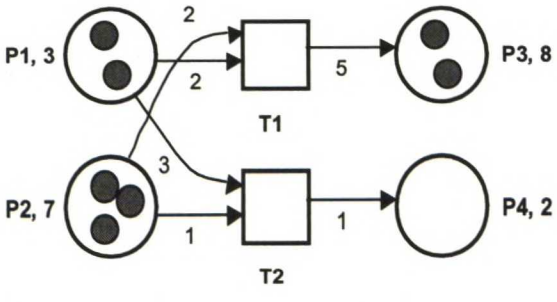
Paikka voi sisältää *merkkejä*[1] tai olla tyhjä. Merkkejä kuvataan yleensä graafisesti paikkaympyrän sisällä olevilla pisteillä. Järjestelmän tilaa tietyllä hetkellä kuvaa kunkin paikan sisältämien merkkien määrä eli verkon *merkintä* M. Ennen järjestelmän käynnistymistä verkolla on *alkumerkintä* M_0.

Jokaiselle paikalle voidaan määritellä tietty *kapasiteetti* kuvauksella $K : S \rightarrow \mathbf{N} \cup \{\omega\}$ [2]. Kapasiteetti ilmaisee suurimman määrän merkkejä, jonka ko. paikka voi sisältää. Tämä merkitään graafisessa esityksessä paikkaympyrän viereen paikan nimen yhteyteen (Kuva 1). Usein kapasiteetti on kuitenkin rajoittamaton, mille käytetään symbolia ω.

*Kuva 1: Paikka **P1**, jonka kapasiteetti on 8 ja merkintä on 5.*

Jokaiseen kaareen liitetään vielä positiivinen kokonaisluku, *kaaripaino*, joka graafisessa esityksessä merkitään kaaren yhteyteen. Kaaripainon määrää kuvaus $W : F \rightarrow \mathbf{N}$.

Tietyn transition sanotaan olevan *vireessä*, jos sen *laukaisuehto* täyttyy, mikä tarkoittaa sitä, että sen jokaisessa esipaikassa on vähintään vastaavan kaaren kaaripainon ilmaisema määrä merkkejä ja jokaisessa jälkipaikassa on tilaa lisätä merkkejä sitä vastaavan kaaren kaaripainon ilmaisema määrä.



*Kuva 2: Transitio **T1** on vireessä, mutta **T2** ei ole.*

Järjestelmä siirtyy tilasta M toiseen tilaan M', kun jokin vireessä oleva transitio t laukeaa. Tällöin siis verkon merkintä muuttuu, mikä ilmaistaan yleensä käyttäen seuraavaa merkintätapaa:

$$M \ [t\rangle \ M'$$

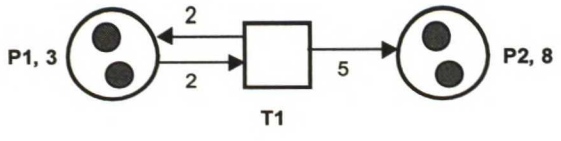
[1] engl. token

[2] $\mathbf{N}$ on luonnollisten lukujen joukko. Tässä esityksessä $0 \notin \mathbf{N}$.

Huomattakoon, että mielivaltainen määrä transitioita voi olla samanaikaisesti vireessä, jolloin niiden sanotaan olevan *rinnakkaisia* ja silloin myös yksi tai useampia niistä voi laueta samanaikaisesti. Jos rinnakkaiset transitiot jakavat esipaikkoja, on kyseessä *kilpailutilanne* eli *konflikti*, joka muuttuu ja viimein purkautuu, kun yksi tai useampi siihen osallistuvista vireessä olevista transitioista laukeaa. Jos yksikään transitio ei ole vireessä, sanotaan järjestelmän olevan *lukkiutunut*[1], ts. se ei voi enää vaihtaa tilaansa.

Transition lauetessa sen jokaisesta esipaikasta poistetaan vastaavan kaaren kaaripainon osoittama määrä merkkejä ja jokaiseen jälkipaikkaan lisätään sitä vastaavan kaaren kaaripainon osoittama määrä merkkejä. On erityisesti huomattava, että kyseessä ei ole merkkien siirtyminen esipaikoista jälkipaikkoihin.

Jokin verkon paikka voi olla tietylle transitiolle samalla sekä esi- että jälkipaikka, jolloin kyseessä on ns. *tiukka silmukka* (Kuva 3). Tällöin edellä määritelty laukaisuehto ei tarkkaan ottaen voi useimmiten toteutua, vaikka niin olisi järjestelmää mallinnettaessa tarkoitettukin. Laukaisuehto voidaan kuitenkin tulkita ikään kuin vaiheittain siten, että ko. paikkaa käsitellään ensin esipaikkana ja poistetaan esikaaren kaaripainon verran merkkejä, laukaistaan transitio ja lisätään jälkikaaren kaaripainon verran merkkejä, jos tilaa on syntynyt kapasiteettiin nähden riittävästi em. poiston jälkeen.



*Kuva 3: Tiukka silmukka paikan **P1** ja transition **T1** välillä. **T1** on vireessä.*

P/T-verkon sanotaan olevan *k-turvallinen*, jos sen jokaisen paikan kapasiteetti ja merkintä missä hyvänsä alkumerkinnästä saavutettavissa olevassa tilassa on korkeintaan k. Erityisesti, jos $k = 1$, sanotaan verkon olevan *1-turvallinen*.

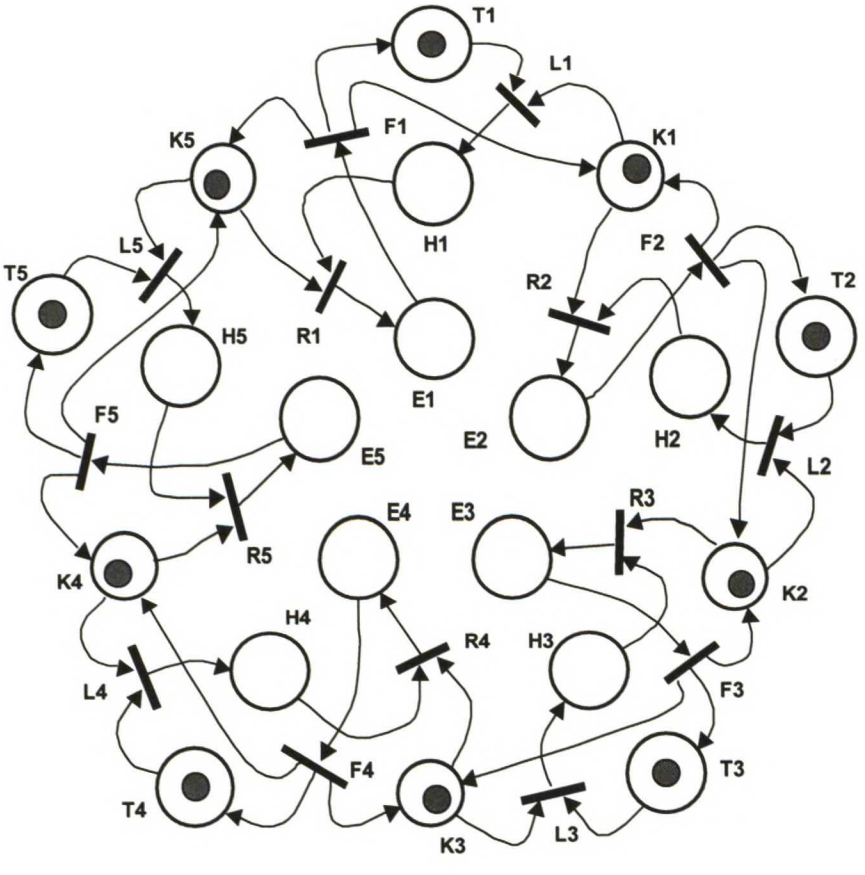
Kuva 4 esittää 1-turvallisena P/T-verkkona kirjallisuudessa usein esiintyvän klassisen[2] esimerkkijärjestelmän, joka kuvaa viiden aterioivan filosofin ongelmaa. Tähän esimerkkiin tullaan myöhemmin palaamaan useissa yhteyksissä.

Ongelman kuvaus on seuraavanlainen: Pyöreän pöydän ympärillä on viisi filosofia. Pöydällä on kunkin edessä lautasellinen spagettia ja aina kahden lautasen välissä haarukka. Syödäkseen filosofi tarvitsee molemmat lautasensa vieressä olevat haarukat. Kukin filosofi voi miettiä, olla nälkäinen tai syödä. Tullessaan nälkäiseksi hän ottaa ensin vasemmalla puolellaan olevan haarukan edellyttäen, että se on vapaana. Otettuaan vasemman haarukan hänen on saatava myös oikeanpuoleinen haarukka, minkä jälkeen hän voi syödä. Syötyään hän palauttaa

[1] engl. deadlocked
[2] Ongelman on alunperin esittänyt E. W. Dijkstra.

molemmat haarukat ja ryhtyy jälleen miettimään. Nälkäisen filosofin on aina syötävä ennen ryhtymistään miettimään. Alkutilanteessa kaikki filosofit miettivät.

Kuvassa paikat on nimetty seuraavasti ($n \in \{1, 2, 3, 4, 5\}$) : **T*n*** = filosofi *n* miettii, **H*n*** = filosofi *n* on nälkäinen, **E*n*** = filosofi *n* syö ja **K*n*** = filosofin *n* *vasemmalla* puolella on vapaa haarukka. Transitiot on nimetty seuraavasti: **L*n*** = filosofi *n* ottaa vasemmalla puolellaan olevan haarukan, **R*n*** = filosofi *n* ottaa oikealla puolellaan olevan haarukan ja **F*n*** = filosofi *n* lopettaa syömisen.



Kuva 4: Viiden aterioivan filosofin ongelma P/T-verkkona.

Esimerkin P/T-verkossa on neljä paikkaa ja kolme transitiota kutakin filosofia kohti. Voidaan helposti havaita, että järjestelmä lukkiutuu, jos jokainen filosofi ottaa vasemmalla puolellaan olevan haarukan, jolloin kukaan ei voi syödä eikä palata miettimään. Verkosta selviää myös nopeasti se ilmeinen ominaisuus, että korkeintaan kaksi, muttei kaksi vierekkäistä filosofia voi syödä yhtäaikaa. Näiden ominaisuuksien tietokoneavusteista verifiointia tarkastellaan myöhemmin kappaleessa 3.4 analysointityökalu MARIA:n avulla.

2.1.2. Predikaatti/transitio-verkot

Edellä kuvattu matalan tason Petri-verkkoesitys muodostuu liian suureksi mallinnettaessa käytännössä esiintyviä laajoja rinnakkaisia järjestelmiä. Koska paikoissa olevat merkit ovat täysin identiteetittömiä, ainoa keino kohdistaa jokin tapahtuma tiettyyn resurssiin on omistaa sille oma esipaikka-transitio-jälkipaikka-ketju, jossa paikkojen kapasiteetit $K(s) = 1$ ja kaarien painot $W(e) = 1$. Jos järjestelmässä esiintyy samankaltaisia resursseja, jotka kuitenkin halutaan erottaa toisistaan, P/T-verkko kasvaa helposti erittäin suureksi, mutta sisältää ko. resurssien lukumäärää vastaavan määrän keskenään samanlaisia rakenteita, kuten helposti havaitaan aterioivien filosofien esimerkistä.

Korkean tason verkoissa rinnakkaiset rakenteet yhdistetään ja merkeille annetaan identiteetit. Tätä kutsutaan verkon *laskostamiseksi*[1].

Predikaatti/transitio-verkot esiteltiin alun perin vuonna 1981 , jolloin H. J. Genrich ja K. Lautenbach määrittelivät niiden matemaattisen formalismin. Myöhemmin tämä formalismi on esitetty uudelleen useissa julkaisuissa, mm. viitteessä [Gen87].

Pr/T-verkossa merkki esitetään *monikkona*[2], jonka alkiot kuuluvat johonkin tiettyyn pohjana olevaan algebralliseen rakenteeseen, ns. *tukeen*[3]. Tässä esityksessä alkiot ovat kokonaislukuja. Esimerkkejä ovat yksikkö < 7 >, kaksikko < 2, 5 >, kolmikko < 120, 45, 0 > jne., sekä edelleen käytettävissä oleva (P/T-verkon) identiteetitön merkki <> eli tyhjä monikko.

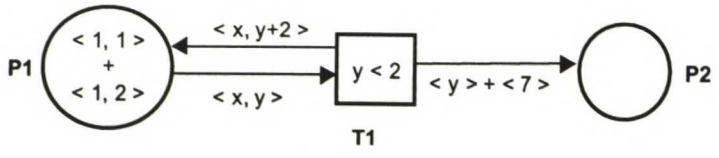
Paikan *ariteetti* on sen sisältämien merkkien eli monikkojen sallittu leveys, eli kaikkien ko. paikassa mahdollisesti esiintyvien merkkien on oltava samaa leveyttä. Merkki poistetaan paikasta tai lisätään sinne aina kokonaisuudessaan. Sama merkki voi esiintyä paikassa useamman kuin yhden kerran, ts. paikka voidaan tulkita *monijoukoksi* tai *koriksi*[4].

Kaaripainoja vastaavat Pr/T-verkossa *kaarilausekkeet* ja transitioon liittyvät *laukaisuehtolausekkeet*, jotka yhdessä määräävät sen, mitä erityisiä monikkoja esipaikoista tulee löytyä ja edelleen, mitä monikkoja jälkipaikkoihin on voitava sijoittaa, jotta transitio olisi vireessä. Huomattakoon, että esi- ja jälkipaikkojen ariteettien ei tarvitse olla samoja. Jokaisen kaarilausekkeen on kuitenkin oltava muodoltaan formaali summa monikoista, joiden leveys vastaa sen paikan ariteettia, johon ko. kaari osuu (poikkeuksiakin sallitaan joskus).

Kaarilausekkeissa voi esiintyä vakioita tai muuttujia, myös niiden funktioita. Tarkasteltaessa transition laukaisuehtoa esikaarten muuttujat sidotaan kyseisten esipaikkojen sisältämiin merkkeihin, kuhunkin vuorollaan. Seuraavan kuvan

[1] engl. folding, vastakohta unfolding.
[2] engl. tuple
[3] engl. support
[4] engl. multiset, bag

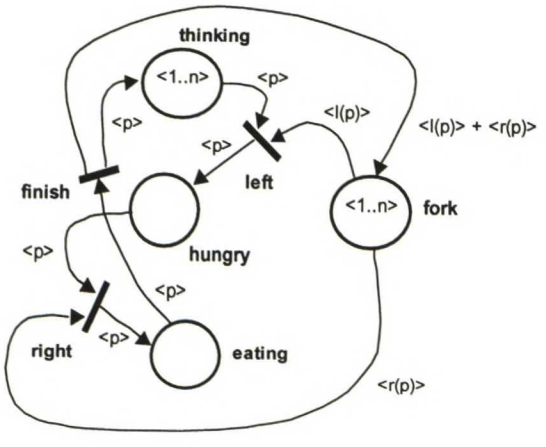
esimerkissä transitio **T1** on vireessä sidonnalla <x,y> := <1,1>, mutta ei sidonnalla <x,y> := <1,2>.



*Kuva 5: Esimerkki Pr/T-verkosta. Transitio **T1** on vireessä.*

Kuvan esimerkistä nähdään, että paikan **P1** ariteetti on kaksi ja paikan **P2** ariteetti on yksi. Ariteettia ei yleensä tarvitse merkitä kaavioon, koska se ilmenee kaarilausekkeista. Paikan **P1** alkumerkintä on < 1, 1 > + < 1, 2 > , mikä voidaan esittää myös muodossa < 1, 1..2 >. Transition **T1** lauettua paikan **P1** merkintä on < 1, 2 > + < 1, 3 > ja paikan **P2** merkintä on < 1 > + < 7 >, eikä **T1** ole enää vireessä, joten järjestelmä on lukkiutunut.

Tarkastellaan edellä ollutta filosofiesimerkkiä Pr/T-verkkona:



Kuva 6: Aterioivien filosofien ongelma Pr/T-verkkona.

Edellä esitetyn mukaisesti vain kutakin filosofia vastaavat neljä paikkaa ja kolme transitiota riittää, kun paikoissa olevat merkit identifioivat tarkasti, mistä filosofista ja haarukasta on kyse. Enää ei verkon rakenne riipu lainkaan siitä, montako filosofia aterialle osallistuu, kunhan haarukoita on sama lukumäärä.

Kaarilausekkeissa esiintyvät funktiot määritellään seuraavasti:

- vasemmanpuoleinen haarukka: $l(x) = x$
- oikeanpuoleinen haarukka: $r(x) = x \bmod n + 1$

Esimerkkiverkossa jokaisen paikan ariteetti on yksi ja kaarilausekkeet ovat siis yksiköiden formaaleja summia. Edellä mainittu lukkiutuma syntyy, kun transitio

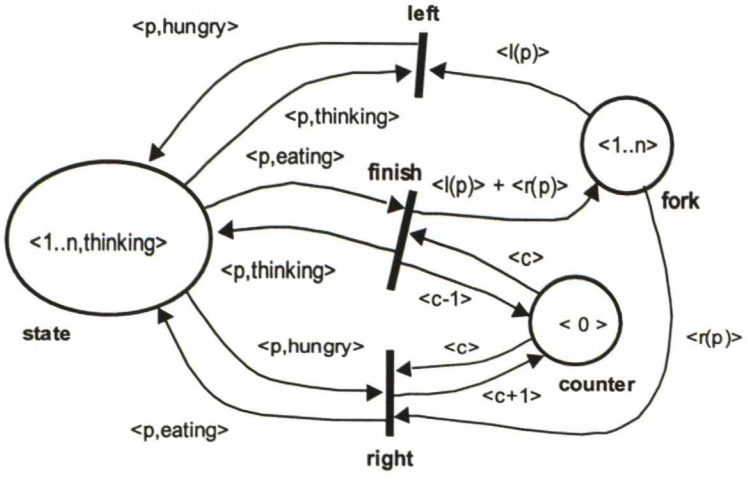
left laukeaa viidesti, jolloin jokaista filosofia vastaava merkki on paikassa **hungry** ja kaikki muut paikat ovat tyhjiä.

2.1.3. Algebralliset järjestelmäverkot

W. Reisig esitteli *algebralliset Petri-verkot* alun perin vuonna 1991 viitteessä [Rei91]. E. Kindler ja H. Völzer täydensivät myöhemmin tätä mallia sallimalla ns. *joustavat kaarilausekkeet*, jolloin alettiin puhua *algebrallisista järjestelmäverkoista*. Tämä formalismi on esitetty mm. viitteessä [KV98].

Oleellisin algebrallisten järjestelmäverkkojen ero edellä esitettyihin Pr/T-verkkoihin on tämän työn kannalta se, että merkit voivat olla teoriassa mielivaltaista algebrallista tyyppiä, jolle on määritelty tietyt algebralliset operaatiot. Tässä tekstissä rajoitutaan tarkastelemaan *tietueita*[1], joiden alkiot voivat olla muutamaa erilaista alkeistyyppiä.

Seuraavassa kuvassa on eräs tapa[2] esittää edellä ollut filosofiesimerkki algebrallisena järjestelmäverkkona:



Kuva 7: Aterioivien filosofien ongelma algebrallisena järjestelmäverkkona.

Nyt kutakin filosofia vastaa kaksi paikkaa ja kolme transitiota. Paikassa **state** olevat merkit ovat tietueita, joiden alkioista toinen on rajoitettua kokonaislukutyyppiä ja toinen kolmearvoista lueteltua tyyppiä (thinking, hungry tai eating). Merkit yksilöivät siis tarkasti, mitä kukin filosofi on tekemässä. Paikassa **fork** olevat merkit kertovat mitkä haarukat ovat vapaana. Kaarilausekkeissa esiintyvät funktiot ovat periaatteessa samat, kuin edellä kappaleessa 2.1.2. Nytkään ei verkon rakenne riipu lainkaan siitä, montako filosofia aterialle osallistuu, kunhan haarukoita on sama lukumäärä. Paikka **counter** on lisätty tiettyjen ominaisuuksien verifioinnin helpottamiseksi (ks. kappale 3.4).

[1] engl. structure
[2] Tässä esitetty ratkaisu on mukailtu MARIA:n esimerkkikokoelmasta [MAR-W].

2.2. Saavutettavuusgraafi ja -analyysi

Petri-verkkomallissa jokainen transition laukeaminen siirtää järjestelmän tilasta toiseen, jolloin tiettyjen paikkojen ja siten koko järjestelmän merkintä muuttuu. Ottamalla alkumerkintää vastaava tila alkutilaksi ja ratkaisemalla kaikki mahdolliset yksittäiset virittyneen transition laukeamiset saadaan alkumerkinnästä *suoraan saavutettavissa* olevat tilat ja niitä vastaavat merkinnät. Kustakin eri merkinnästä edelleen jatkaen saadaan lopulta aikaan tilakaavio, jota nimitetään *saavutettavuusgraafiksi*.

2.2.1. Saavutettavuusgraafi

Saavutettavuusgraafin $N = (V, E)$ kaksi pääkomponenttia ovat

1. *Tila* eli *merkintä* (solmu) $M \in V$.
2. *Kaari* $(M,t,M') \in E$. $E \subseteq (V \times T \times V)$.

Lisäksi pätee:

3. $M_0 \in V$.
4. $M \in V \wedge t \in T \wedge M\ [t > M' \Rightarrow M' \in V \wedge (M,t,M') \in E$.
5. V:ssä ja E:ssä ei ole muita alkioita.

V on tilojen eli merkintöjen ja E kaarten joukko, joille pätee: $V \cap E = \varnothing$. Huomattakoon, että saavutettavuusgraafi voi sisältää ns. *terminaalisolmuja*, joista ei lähde yhtään kaarta. Nämä ilmaisevat järjestelmässä olevan lukkiutumia. Lisäksi graafit sisältävät usein ei-triviaaleja *vahvasti kytkettyjä komponentteja*, jotka myös ilmaisevat tutkijaa kiinnostavia järjestelmän ominaisuuksia.

2.2.2. Saavutettavuusanalyysi

Saavutettavuusgraafi sisältää siis kaikki alkumerkinnästä saavutettavissa olevat tilat, joten sen kattava läpikäyminen paljastaisi kaikki sellaiset tilat, joista tutkija mahdollisesti olisi kiinnostunut virheiden tms. etsimiseksi. Käytännössä kuitenkin saavutettavuusgraafin koko usein kasvaa räjähdysmäisesti järjestelmän pienenkin kasvun tai rakennemuutoksen seurauksena, eikä sen kattava tutkiminen ole mahdollista. Kuten myöhemmin kappaleessa 3.4 tulemme huomaamaan, viiden filosofin Petri-verkkomallin saavutettavuusgraafi sisältää 82 tilaa ja 265 kaarta niiden välillä ja kun filosofien lukumäärää kasvatetaan kahdeksaan, saadaan 1154 tilaa ja 5968 kaarta. Pienestäkin verkosta muodostuu siis helposti käytännöllisesti katsoen liian suuri graafi.

Erityisen suureksi ongelmaksi saavutettavuusgraafin koko muodostuu siksi, että kokoa ei yleensä edes pystytä arvioimaan etukäteen riittävän tarkasti, vaan vasta graafia muodostettaessa joudutaan toteamaan sen vievän liian paljon aikaa tai tietokoneen muistikapasiteettia ja koko operaatio joudutaan keskeyttämään il-

man hyödyllistä lopputulosta.

Useita erilaisia menetelmiä on kehitetty saavutettavuusgraafin koon pienentämiseksi jättämällä osa tiloista muodostamatta, koska niiden merkitys tietyn kiinnostavan lopputuloksen kannalta on merkityksetön. Näistä voidaan mainita erityisesti *symmetriamenetelmä* ja *itsepäisten joukkojen*[1] *menetelmä*. Tässä tekstissä ei kuitenkaan enempää käsitellä näiden, eikä muidenkaan saavutettavuusgraafin rajoittamismenetelmien käyttöä.

Saavutettavuusanalyysiä voidaan kuitenkin tehdä menestyksellisesti myös generoimatta lainkaan koko saavutettavuusgraafia. Nykyaikaiset analyysityökalut tarjoavat mahdollisuuden *simuloida* transitioiden laukeamisia yksi kerrallaan, jolloin tutkija voi keskittyä seuraamaan vain erityisen kiinnostavia polkuja tai tarkastella, miten malli haarautuu eri tiloista.

2.3. Lineaarinen temporaalilogiikka ja mallintarkastus

Välimuoto edellisessä kappaleessa kuvatuille koko saavutettavuusgraafin generoinnille ja askeleittain simuloinnille on *mallintarkastus*[2] verifioimalla tiettyjen tilanteiden esiintyminen tai esiintymättömyys järjestelmän suorituksen aikana.

2.3.1. LTL

Tietyn tilanteen esittämiseen tilasekvenssin suorituksen aikana tarvitaan kieli. *Lineaarinen temporaalilogiikka LTL*[3] on eräs *modaalilogiikan* laji, mikä tarkoittaa sitä, että se laajentaa staattisen logiikan koskemaan järjestelmän tilojen välisiä suhteita suorituksen kuluessa. LTL:ssa lineaarisuus tarkoittaa, että jokaisella tilalla on yksikäsitteisesti määritelty tulevaisuus, ts. haarautumista ei tapahdu. Seuraava esitys perustuu pääasiassa lähteeseen [Pel01].

Olkoon $\mathcal{U}$ alla oleva staattinen logiikka eli atomisten propositioiden joukko. LTL:n syntaksi määritellään seuraavasti:

1. Kaikki $\mathcal{U}$:n kaavat ovat myös LTL:n kaavoja.
2. Jos φ ja ψ ovat kaavoja, niin myös $(\neg\varphi)$, $(\varphi \wedge \psi)$, $(\varphi \vee \psi)$, $(\bigcirc\varphi)$, $(\Diamond\varphi)$, $(\Box\varphi)$, $(\varphi \mathbf{U} \psi)$ ja $(\varphi \mathbf{V} \psi)$ ovat kaavoja.
3. Muita LTL-kaavoja kuin kohdissa 1 ja 2 mainitut ei ole.

LTL-kaava tulkitaan päättymättömän tilasekvenssin $\xi = x_0 x_1 x_2 \ldots$ yli. Merkitään ξ^k:lla osasekvenssiä, joka alkaa tilasta x_k ja käsittää kaikki sen jälkeiset tilat. LTL:n semantiikka voidaan määritellä seuraavasti:

[1] engl. stubborn sets
[2] engl. on-the-fly verification
[3] engl. linear temporal logic

4. $\xi^k \models \eta \Leftrightarrow x_k \models \eta$, missä $\eta \in \mathcal{U}$.
5. $\xi^k \models (\neg\varphi) \Leftrightarrow \xi^k \not\models \varphi$.
6. $\xi^k \models (\varphi \wedge \psi) \Leftrightarrow \xi^k \models \varphi \wedge \xi^k \models \psi$.
7. $\xi^k \models (\varphi \vee \psi) \Leftrightarrow \xi^k \models \varphi \vee \xi^k \models \psi$.
8. $\xi^k \models (\bigcirc\varphi) \Leftrightarrow \xi^{k+1} \models \varphi$.
9. $\xi^k \models (\Diamond\varphi) \Leftrightarrow \exists i \geq k$ siten, että $\xi^i \models \varphi$.
10. $\xi^k \models (\Box\varphi) \Leftrightarrow \forall i \geq k : \xi^i \models \varphi$.
11. $\xi^k \models (\varphi \mathbf{U} \psi) \Leftrightarrow \exists i \geq k$ siten, että $\xi^i \models \psi$ ja $\forall j \in [k, i) : \xi^j \models \varphi$.
12. $\xi^k \models (\varphi \mathbf{V} \psi) \Leftrightarrow \forall i \geq k : \xi^i \models \psi$ tai $\exists j \geq k$ siten, että $\xi^j \models \varphi$ ja $\forall i \in [k, j] : \xi^i \models \psi$.

Sääntö 4 ilmaisee, että alla olevan logiikan $\mathcal{U}$ kaava tulkitaan osasekvenssin ξ^k ensimmäisessä tilassa x_k.

Selväkielisesti ilmaistuna $(\bigcirc\varphi)$ tarkoittaa, että kaava φ toteutuu tilaa k seuraavassa tilassa $k + 1$, $(\Diamond\varphi)$ tarkoittaa, että kaava φ lopulta toteutuu joskus tulevaisuudessa ainakin yhdessä tilassa, $(\Box\varphi)$ tarkoittaa, että kaava φ toteutuu tilasta k lähtien jokaisessa tilassa. Binäärinen operaattori $(\varphi \mathbf{U} \psi)$ ilmaisee, että kaava φ toteutuu jokaisessa tilassa ainakin sitä tilaa edeltävään tilaan saakka, jossa kaava ψ toteutuu ja $(\varphi \mathbf{V} \psi)$ ilmaisee, että kaava ψ toteutuu ainakin vielä siinä tilassa, jossa kaava φ toteutuu, mutta sen jälkeen sen ei tarvitse toteutua, ts. kaavan φ toteutuminen jossain tulevassa tilassa *vapauttaa* kaavan ψ.

2.3.2. Mallintarkastus analyysityökaluilla

Nykyaikaisilla analyysityökaluilla on mahdollisuus generoida saavutettavuusgraafi siten, että samalla tarkistetaan jatkuvasti LTL-kaavan voimassaolo jokaisessa generoidussa tilassa. Haluttaessa analyysi voidaan keskeyttää ja näin useimmiten tehdäänkin halutun ehdon täyttyessä, jolloin suoritusaikaa voidaan rajoittaa.

LTL-kaava annetaan analysaattorille tietyn syntaksin mukaan joko mallitiedostossa tai komentorivillä. Tässä työssä perehdytään lähemmin ainoastaan MARIA-analysaattorin tapaan tehdä mallintarkastusta, koska se on erityisesti vaikuttanut suunnitellun käyttöliittymän ominaisuuksiin.

2.4. Kuvauskieli SDL

Tietoliikennesovellukset ja niissä käytetyt tiedonsiirtoprotokollat muodostavat erään tärkeimmistä rinnakkaisten ja hajautettujen järjestelmien tutkimuksen osa-alueista. Sovellukset kuvataan formaalisti käyttäen esimerkiksi SDL[1]-kieltä, joka on yleisyytensä ja soveltuvuutensa ansiosta valittu ensimmäiseksi kohteeksi

[1] specification and description language

tutkittaessa käytännön esityskielten käyttöä analysaattoreiden lähdekielenä. Tässä kappaleessa luodaan lyhyt katsaus SDL-kielen kehitykseen ja yleispiirteisiin. Laajempi esitys on esimerkiksi viitteessä [EHS97].

Alkuperäisestä CCITT:n[1] 1970-luvulla epäformaalisti määrittelemästä graafisesta esitystavasta SDL on useiden vaiheiden kautta kehittynyt nykyiseen ITU-T:n[2] suosituksessa Z.100 [ITU99] formaalisti määriteltyyn kieleen. SDL-kieltä on jo pitkään käytetty laajalti sekä määrittelykielenä että myös automaattisten ohjelmakoodin generaattorien lähdekielenä.

SDL:stä on määritelty kaksi versiota. Graafinen versio SDL/GR[3] koostuu pääasiassa graafisista symboleista, joiden avulla esitetään järjestelmän kuvaus kaavioina. Tekstipohjainen versio SDL/PR[4] määrittelee puhtaasti sanallisiin fraaseihin perustuvan, hierarkkisen tavan kuvata järjestelmiä. Graafinen esitys on havainnollinen järjestelmiä suunniteltaessa ja kuvattaessa muille henkilöille esitettäväksi, mutta ainoastaan tekstipohjainen versio kelpaa lähtökohdaksi automaattiseen kääntämiseen ja analyysiin.

2.4.1. Järjestelmän SDL-kuvaus

SDL-kuvauksessa *järjestelmä* (SYSTEM) koostuu *prosesseista* (PROCESS), *proseduureista* (PROCEDURE) ja *lohkoista* (BLOCK). Lohkoja käytetään kuvauksen modulaarisuuden parantamiseen kokoamalla esimerkiksi tietyn tyyppisiä tehtäviä yhteen ja ne koostuvat prosesseista ja mahdollisesti edelleen muista lohkoista. Proseduurit ovat prosessien sisäisiä aliohjelmia, joita prosessi kutsuu (CALL) ja jotka päättyvät RETURN-lauseeseen. Järjestelmällä on *ympäristö*, josta se saa herätteitä, mutta jonka sisäisestä toiminnasta sen ei tarvitse välittää.

Prosessit ovat toiminnaltaan rinnakkaisia ja kommunikoivat keskenään asynkronisesti lähettämällä toisilleen *viestejä*. Kukin prosessi on tilakone, joka mahdollisesti vaihtaa tilaansa (NEXTSTATE) saatuaan (INPUT) viestin ja suoritettuaan *transition*. Tätä siirtymää tilojen välillä nimitetään jatkossa SDL-transitioksi erotukseksi Petri-verkon transitiosta samoin kuin järjestelmän tilaa SDL-tilaksi erotukseksi saavutettavuusgraafin tilasta. SDL-transitiossa prosessi voi muuttaa muuttujien arvoja sekä lähettää (OUTPUT) viestejä.

Prosessin vastaanottamat viestit säilytetään *jonossa*, jonka pituus on SDL:n määritelmän mukaan ääretön, mutta käytännössä aina rajallinen. Ollessaan sopivassa tilassa prosessi poistaa jonosta vanhimman viestin eli *kuluttaa* sen ja suorittaa siihen liittyvän SDL-transition. Viestien järjestystä jonossa voidaan muuttaa SAVE-toiminnolla. Jos viestiin ei liity SDL-transitiota, se pelkästään poistetaan jonosta, mitä sanotaan *implisiittiseksi kulutukseksi*.

---

[1] Comité Consultatif Internationale de Telégraphique et Teléphonique
[2] ITU-T on ITU:n (International Telecommunication Union) tietoliikenteen standardointisektori
[3] graphic representation
[4] phrase representation

Osa prosesseista luodaan järjestelmän käynnistyessä ja ne ovat ns. *pääprosesseja* (MASTER PROCESS). Kustakin pääprosessista on koko järjestelmän suorituksen ajan olemassa täsmälleen yksi esiintymä. Muut prosessit ovat *aliprosesseja*[1] (PROCESS) ja niitä voidaan luoda (CREATE) dynaamisesti. Aliprosessissa olevassa SDL-transitiossa prosessi voi myös pysäyttää (STOP) itsensä.

Prosessilla on joukko sisäisiä pid-tyyppisiä[2] muuttujia. Muuttuja SELF sisältää prosessin oman pid:in, OFFSPRING sisältää prosessin viimeksi luoman aliprosessin pid:in, PARENT puolestaan sen prosessin pid:in joka loi tämän prosessin ja SENDER sen prosessin pid:in, jonka lähettämä viesti laukaisi nykyisen SDL-transition.

Viestit sisältävät nimensä, lähettäjän pid:in ja mahdollisia parametreja, jotka voivat olla eri tietotyyppejä. Saman nimisissä viesteissä on aina samanlaiset parametrit.

Järjestelmässä voi esiintyä *ajastimia*. Prosessi käynnistää (SET) ajastimen ja vastaanottaa viestin sen lauetessa määritellyn ajan kuluttua. Prosessi voi myös pysäyttää (RESET) ajastimen. Jos ajastin on pysäytettäessä jo ehtinyt lähettää viestin, tämä poistetaan jonosta.

2.4.2. TNSDL

TNSDL[3] on SDL-kielen muunnos, joka on kehitetty Nokia-yhtiön tarpeisiin [LRKT95]. Tässä työssä käytetty ja muutettu EMMA-kääntäjä (kappale 3.3) käyttää lähdekielenään TNSDL:n versiota 3.0, joka puolestaan perustuu SDL88-kieleen [ITU88].

TNSDL sisältää joukon laajennuksia ja toisaalta myös rajoituksia SDL88-kieleen verrattuna. Mitkään näistä eivät kuitenkaan ole oleellisia EMMA:n alkuperäisen toteutuksen kannalta eivätkä siten myöskään vaikuta mitenkään tässä työssä tehtyihin muutoksiin, joten niitä ei tässä käsitellä.

[1] engl. hand process
[2] process identifier
[3] Telenokia SDL. Telenokia on nykyisin Nokia Telecommunications.

3. Analysointityökalut

Tässä luvussa esitellään TKK:n tietojenkäsittelyteorian (aiemmin digitaalitekniikan) laboratoriossa kehitettyjä analysointityökaluja. Mainittakoon, että myös muissa suomalaisissa ja ulkomaisissa korkeakouluissa on tehty vastaavaa tutkimusta ja kehitetty työkaluohjelmistoja, mutta niihin ei tässä esityksessä puututa.

3.1. Historiaa

Järjestelmien automaattinen verifiointi Petri-verkkojen avulla alkoi 1970-luvulla ja 1980-luvun alkupuolella oli jo saatu aikaan analysaattoreita, jotka kykenivät generoimaan joitakin kymmeniä tuhansia tiloja käsittäviä saavutettavuusgraafeja. Viitteen [Mäk00] johdanto-osuudessa viitataan tutkimuksiin, joiden mukaan 1980-luvun lopulla oli olemassa 13 Petri-verkkojen saavutettavuusanalyysiä suorittavaa työkalua, mutta todetaan tilanteen vakiintuneen 1990-luvun lopulla kymmenkuntaan.

Suomessa tietokoneella tapahtuvan saavutettavuusanalyysin tutkimus alkoi 1980-luvun alkupuolella ja eräänä lopputuloksena TKK:n digitaalitekniikan laboratorio julkaisi vuonna 1988 PRENA-nimisen työkalun Pr/T-verkkojen analysointiin. PRENA:n suurin rajoitus oli, että se kykeni generoimaan vain muutamia tuhansia saavutettavia tiloja, mikä mahdollisti vain erittäin yksinkertaisten Petri-verkkomallien käsittelyn.

3.2. PROD

PRENA:n seuraajaksi alettiin TKK:n digitaalitekniikan laboratoriossa kehittää parempaa Pr/T-verkkoanalysaattoria pääasiassa opiskelijavoimin ja vuonna 1991 julkaistiin ensimmäinen versio PROD-analysaattorista [GTV93]. Myöhemmin sitä on kehitetty edelleen useiden opiskelijoiden ja tutkijoiden toimesta ja se kykenee käsittelemään Petri-verkkoja, joiden tila-avaruus käsittää miljoonia saavutettavia tiloja, joten sillä voidaan jo analysoida myös käytännön järjestelmien malleja.

PROD:in käyttämä Petri-verkkojen kuvauskieli [VHHP95] on C-ohjelmointikieli täydennettynä erityisillä direktiiveillä. Verkkokuvaus käännetään suoritettavaksi ohjelmaksi, joka puolestaan generoi saavutettavuusgraafin. Ainoa käytettävissä oleva tietotyyppi on kokonaislukujen monikko.

Seuraava kuva esittää kappaleessa 2.1.2 (Kuva 6, sivu 10) olevan aterioivien filosofien ongelman Pr/T-verkkomallin PROD-kielellä[1]:

[1] Malli on mukailtu viitteestä [VHHP95] muuttaen siten, että se vastaa kappaleessa 3.4 esitettyä MARIA-kielistä mallia nimeämisen ja saavutettavien tilojen lukumäärän osalta.

```
#define n 5
#define l(x) (x)
#define r(x) (1 + ((x) % n))

#place thinking lo(<.1.>) hi(<.n.>) mk(<.1..n.>)

#place fork mk(<.1..n.>)

#place hungry lo(<.1.>) hi(<.n.>)

#place eating lo(<.1.>) hi(<.n.>)

#trans left
   in  { thinking: <.p.>; fork: <.l(p).>; }
   out { hungry: <.p.>; }
#endtr

#trans right
   in  { fork: <.r(p).>; hungry: <.p.>; }
   out { eating: <.p.>; }
#endtr

#trans finish
   in  { eating: <.p.>; }
   out { fork: <.l(p).>+<.r(p).>; thinking: <.p.>; }
#endtr
```

Kuva 8: Aterioivien filosofien ongelma PROD-kielellä.

Seuraavassa esitetään filosofiesimerkin käsittely PROD:illa.

```
~/prod$ ./bin/prod dining.init
~/prod$ ./dining
~/prod$ ./bin/strong dining
~/prod$ ./bin/probe dining
0#statistics
Number of nodes: 82
Number of arrows: 265
Number of terminal nodes: 1
Number of nodes that have been completely processed: 82
Number of strongly connected components: 2
Number of nontrivial terminal strongly connected components: 0
0#quit
~/prod$
```

Esimerkistä nähdään seuraavat vaiheet: käännös `prod`-ohjelmalla, `dining`-ohjelman suoritus, eli saavutettavuusgraafin generointi, vahvasti kytkettyjen komponenttien generointi `strong`-ohjelmalla ja lopuksi tulosten tarkastelu `probe`-ohjelmalla, jonka `statistics`-komennolla on kysytty tietoja saavutettavuusgraafista.

3.3. EMMA

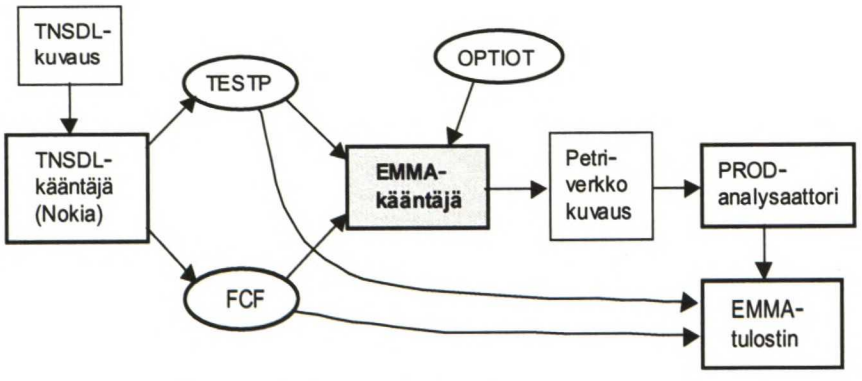
Vuonna 1995 käynnistettiin TKK:n digitaalitekniikan laboratoriossa projekti, jonka päämääränä oli aikaansaada TNSDL-kielellä kuvattujen järjestelmien analyysityökalu EMMA[1]. Aluksi tehtiin laaja esiselvitys [HOPV95] TNSDL:n erilaisten rakenteiden mallintamisesta Petri-verkkojen avulla. Samasta aiheesta

[1] extendible multi-method analyzer

on myöhemmin julkaistu useita projektin aikana saaduilla kokemuksilla täydennettyjä raportteja, mm. [Hus96], [HMJ96], [Hus98], [HM99], [Jyr97] ja [Mal97].

EMMA:n tarkoitus on olla analysaattori, jolle voidaan antaa syötteeksi TNSDL-kielinen esitys järjestelmästä ja joka esittää analyysitulokset sellaisessa muodossa, että yhteys alkuperäiseen TNSDL-esitykseen säilyy. Tässä mielessä sen tulisi sisältää PROD-analysaattori käyttäjälle näkymättömänä osanaan, mutta näin pitkälle integroitua työkalua EMMA:sta ei ole tehty. Toimiva EMMA-*kääntäjä* valmistui vuoden 1997 lopulla ja kehitystyö lopetettiin uuden MARIA-projektin käynnistyessä.

EMMA:n toiminta on esitetty seuraavassa kuvassa:



Kuva 9: EMMA-analysaattorin toiminta.

EMMA ei siis tee Petri-verkkoesitystä suoraan TNSDL-kielestä, vaan ensin joudutaan muodostamaan Nokian omistamaa TNSDL-kääntäjää käyttäen kaksi välitiedostoa TESTP[1] (`.sym`) ja FCF[2] (`.fcf`), jotka puolestaan toimivat syötteinä EMMA:lle. Käännöstä voidaan ohjata optioilla (`.opt`). Käännös tapahtuu komennolla:

```
emma -analyze <tiedoston nimi> > <mallitiedoston nimi>
```

Käännöksestä saatu Petri-verkkomalli analysoidaan PROD:illa, jonka `probe`-ohjelman tulosteita voidaan edelleen syöttää EMMA-*tulostimelle* yhdistettäväksi TNSDL-mallin symboleiden kanssa.

Tässä työssä EMMA-kääntäjää muutettiin siten, että se tuottaa PROD-kielisen mallin sijasta MARIA-kielisen mallin. Muutostyöstä kerrotaan luvussa 5, jossa myös kuvataan ohjelman rakennetta. EMMA-tulostimeen ei tehty muutoksia, koska tulostus perustuu täysin `probe`-ohjelman tulostuksen käyttöön ja olisi siten joka tapauksessa ohjelmoitava täydellisesti uudelleen esimerkiksi käyttöliittymän osana.

[1] TNSDL external symbol table presentation
[2] flow control format

3.4. MARIA

PROD:in tietotyyppien rajallisuus, käyttöliittymän monimutkaisuus ja epämodulaarisuudesta johtuva jatkokehittämisen vaikeutuminen johtivat tarpeeseen kehittää kokonaan uusi ja parempi analysaattori. Varsinkin EMMA-projekti osoitti tarpeelliseksi sisällyttää Petri-verkkomalliin erityinen *jonotyyppi* tiedonsiirtoprotokollien mallintamista varten.

Vuoden 1998 alussa käynnistetty ja vuonna 2001 päättyvä nelivuotinen MARIA[1]-projekti johti täysin uudenlaisen, algebrallisiin järjestelmäverkkoihin perustuvan MARIA-analysaattorin [Mäk00] syntymiseen. MARIA kykenee generoimaan kymmeniä miljoonia tiloja käsittävän saavutettavuusgraafin nopeasti ja käyttäen mahdollisimman vähän tietokoneen muistitilaa. MARIA on myös nimensä mukaisesti modulaarinen, mikä tarkoittaa sitä, että siihen voidaan helposti liittää esikäsittelymoduuleita erilaisille lähdekielille, mm. SDL:lle [Mäk98], samoin kuin uusia menetelmiä käyttäviä analyysimoduuleja. Tätä työtä tehtäessä MARIA:a kehitettiin vielä voimakkaasti ja versio 1.0 julkaistiin työn loppuvaiheessa.

Seuraava kuva esittää kappaleessa 2.1.3 (Kuva 7, sivu 11) olevan aterioivien filosofien ongelman Petri-verkkomallin MARIA-kielellä. Mallitiedostoon on lisätty komennot raportoida lukkiutumatilat ja sellaiset tilat, joissa kolme filosofia on samanaikaisesti syömässä.

```
int N = 5;
typedef unsigned (1..N) phil_t;

typedef struct {
  phil_t p,
  enum { thinking, hungry, eating } s
} stat_t;

place fork (0..#phil_t) phil_t: phil_t p: p;
place counter unsigned: 0;
place state (#phil_t) stat_t: phil_t p: { p, thinking };

trans left
  in  { place state: { p, thinking }; place fork: p; }
  out { place state: { p, hungry }; };

trans right
  in  { place state: { p, hungry }; place fork: +p;
        place counter: c; }
  out { place state: { p, eating }; place counter: c+1; };

trans finish
  in  { place state: { p, eating }; place counter: c; }
  out { place state: { p, thinking }; place fork: p, +p;
        place counter: c-1; };

deadlock true;
reject (3 subset place counter);
```

Kuva 10: Aterioivien filosofien ongelma MARIA-kielellä.

[1] modular reachability analyser

Esimerkin ja sitä jatkossa käsittelevien esimerkkiajojen täydelliseksi ymmärtämiseksi vaadittaisiin, että lukija on perehtynyt käyttäjän käsikirjaan [Mäk01], mutta tämän tekstin seuraaminen ei sitä välttämättä edellytä.

MARIA:n käyttämä Petri-verkkojen kuvauskieli muistuttaa jonkin verran PROD-kieltä, mutta on kuitenkin täysin eri kieli, eikä siinä myöskään käytetä C-kielen direktiivejä samalla tavalla. Myöskään saavutettavuusgraafin generoivaa, erikseen suoritettavaa ohjelmaa ei muodosteta. Käytettävissä olevien tietotyyppien määrä on laaja ja käytännön sovelluksiin suuntautuneesti suunniteltu. Tietotyypit määrittelyineen pohjautuvat läheisesti C-kieleen. Myös esimerkiksi loogisten lausekkeiden syntaksi muistuttaa mahdollisimman paljon C-kieltä.

MARIA-kielessä monet operaatiot ilmaistaan erittäin kompaktissa muodossa, kuten esitetyssä esimerkissä esiintyvä +p, joka tarkoittaa kappaleessa 2.1.2 esitettyä funktiota *r(p)*.

Seuraavassa esitetään filosofiesimerkin käsittely MARIA:lla.

```
~/test$ maria -v -p lbt -b dining.pn
dining.pn:3 places and 3 transitions
dining.pn:computing the initial marking
dining.pn:generating the reachability graph
deadlock state @71
"dining.pn": 82 states, 265 arcs
@0$
```

Esimerkkiajossa generoidaan ensin saavutettavuusgraafi ja suoritetaan mallitiedostossa olevat mallintarkastuslauseet `deadlock` ja `reject`. Mahdollisten LTL-kaavojen käyttö mallintarkastuksessa edellyttää LBT[1]-kaavamuuntimen [MAR-W] käyttöä. MARIA:n tulostuksesta nähdään, että saavutettavuusgraafi vastaa edellä kappaleessa 3.2 PROD:illa generoitua ainakin tilojen ja kaarten lukumäärän osalta. Lukkiutumia on yksi tilassa, jonka tunnus on `@71`. Mallitiedostossa oleva `reject`-lause ei ole aiheuttanut mitään tulostusta, mikä osoittaa, että paikassa **counter** oleva laskuri ei missään tilassa saavuta arvoa kolme ja siten korkeintaan kaksi filosofia voi syödä samanaikaisesti.

Onnistuneen suorituksen päätteeksi MARIA jää odottamaan käyttäjän komentoja interaktiivisessa tilassa. Tässä on vielä esimerkin vuoksi verifioitu `eval`-komennolla ja LTL-kaavalla, että edellä mainittu ehto pätee aina:

```
@0$eval []!(3 subset place counter)
dining.pn:translating the formula
dining.pn:performing model checking
(command line):2:property holds
@0$
```

[1] LTL to Büchi automaton translator

Lopuksi katsotaan lukkiutumatilan @71 merkintä ja päätetään MARIA-istunto:

```
@0$show @71
deadlock state (
 counter:
  0
 state:
  {1,hungry},{2,hungry},{3,hungry},{4,hungry},{5,hungry}
)
5 predecessors
@0$exit
~/test$
```

Jos halutaan verifioida, että todella kaksi filosofia syö joskus samanaikaisesti, voidaan viimeisen rivin kaava muuttaa muotoon

```
reject (2 subset place counter);
```

jolloin saadaan luettelo kaikista niistä tiloista, joissa näin tapahtuu. Muutoksen tulokset ilmenevät seuraavasta esimerkkiajosta:

```
~/test$ maria -p lbt -b dining.pn
rejected state @58
rejected state @60
rejected state @67
rejected state @68
rejected state @70
rejected state @75
rejected state @77
rejected state @78
rejected state @80
rejected state @81
deadlock state @71
"dining.pn": 82 states, 265 arcs
@0$
```

Nähdään, että kaikkiaan kymmenessä tilassa kaksi filosofia syö samanaikaisesti. Jos halutaan verifioida, että kaksi *vierekkäistä* filosofia ei voi syödä samanaikaisesti, voidaan mallitiedostoon sisällyttää `reject`-lauseet, joissa suljetaan yksi kerrallaan pois kaikki vierekkäisten filosofien muodostamien parien esiintymiset syömässä, mutta parametrin N muuttuessa lauseita jouduttaisiinn lisäämään tai poistamaan. Myös interaktiivisessa tilassa tämä voidaan tehdä. Seuraavassa on verifioitu, että filosofeille 1 ja 2 ehto pätee:

```
@0$eval []!(is stat_t {1,eating} union is stat_t \
@0>{2,eating} subset place state)
dining.pn:translating the formula
dining.pn:performing model checking
(command line):2:property holds
@0$
```

Yllä on käytetty jatkorivimerkkiä "\", jolloin seuraava rivi kuuluu samaan komentoon.

Edellä esitettyjen tapojen lisäksi ehkä suoraviivaisimmaksi tavaksi jää katsoa kunkin luetellun tilan merkintä vuorotellen ja tarkistaa, mitkä filosofit ovat syömässä. Seuraavassa on katsottu `show`-komennolla tila `@80`:

```
@0$hide counter; show @80
state (
 state:
  {1,thinking},{4,thinking},{2,hungry},{3,eating},{5,eating}
)
3 predecessors
2 successors
@0$exit
~/test$
```

Lopuksi tarkastellaan vielä saavutettavuusgraafin kasvua filosofien lukumäärää kasvattamalla. Seuraavissa esimerkkiajoissa generoidaan saavutettavuusgraafit mallitiedosteoille `din`*n*`.pn`, missä *n* kertoo filosofien lukumäärän:

```
~/test$ maria -b din6.pn -e exit
"din6.pn": 198 states, 768 arcs
~/test$ maria -b din7.pn -e exit
"din7.pn": 478 states, 2163 arcs
~/test$ maria -b din8.pn -e exit
"din8.pn": 1154 states, 5968 arcs
~/test$
```

Edellä olleet esimerkkiajot kuvaavat lyhyesti MARIA:n tuomaa selkeää parannusta PROD:iin verrattuna niin kuvauskielen ilmaisuvoimaisuuden kuin käytettävyydenkin osalta. Niistä ilmenee kuitenkin myös komentorivipohjaisuuden epäkäytännöllisyys, mikä edelleen korostuu otettaessa käyttöön pitempiä muuttujien, paikkojen ja transitioiden nimiä.

Tottunut MARIA:n käyttäjä löytää pian keinoja kiertää komentorivipohjaisessa käytössä usein esiintyvät, pitkien komentorivien ja näyttöruudulta karkaavien tulostusten aiheuttamat hankaluudet, mutta aloittelevalle käyttäjälle ne saattavat tuottaa niin paljon vaivaa, että ohjelman käyttö tuntuu tarpeettoman työläältä. Luvussa 7 palataan vielä filosofiesimerkin käsittelyyn MARIA:lla graafisen XMARIA-käyttöliittymän avulla, jolloin havaitaan, kuinka mainitut hankaluudet voidaan suurimmaksi osaksi ratkaista.

4. Tietokoneohjelmien käyttöliittymät

Tässä luvussa esitellään nykyaikaisten graafisten käyttöliittymien perusperiaatteita ja tarkastellaan erityisesti sitä, miten käyttöliittymä saadaan sellaiseksi, että käyttäjän on helpointa omaksua sekä itse käyttöliittymän että taustalla olevan tiedon käyttö. Esityksen pohjana on käytetty pääasiassa lähteitä [Kal95], [Kal92] ja [Pre94].

4.1. Graafiset käyttöliittymät

Nykypäivän tietokoneteknologia on niin pitkälle kehittynyt, että yleisesti pidetään itsestään selvänä, että käyttöliittymä on graafinen.

Termillä *graafinen käyttöliittymä* tarkoitetaan tässä, kuten hyvin yleisesti muussakin kirjallisuudessa ja nykyisessä tietotekniikan kielenkäytössä sellaista käyttöliittymää, jolla tietokonetta tai tiettyä ohjelmistoa käytetään pääasiallisesti hiiren tai muun vastaavan osoitinlaitteen avulla, tarvitsematta juuri kirjoittaa mitään komentoja.

Graafisuuden ei siis tarvitse tarkoittaa itse tiedon esittämistä kuvien avulla, mikä olisikin epäoleellinen vaatimus esimerkiksi tekstinkäsittelyohjelmalle. Yleinen harhakäsitys on myös se, että graafinen käyttöliittymä aina sisältää kauniita graafisia elementtejä, kuten ikoneja, jopa animaatioita. Näin ei suinkaan tarvitse olla, vaan jo hyvin yksinkertaisella ja pelkistetyllä grafiikalla saadaan aikaan tavoiteltu helppokäyttöisyys.

4.2. Ihmisen ja tietokoneen vuorovaikutus tutkimusalueena

Tietokoneiden tultua osaksi lähes jokaisen ihmisen päivittäistä elämää on tullut tarpeelliseksi ryhtyä kiinnittämään huomiota niiden mahdollisimman helppoon ja yhdenmukaiseen käytettävyyteen. *Ihmisen ja tietokoneen vuorovaikutus*[1] on vakiintunut ilmaus, joka kattaa erittäin laajan ja paljon tutkitun tietojenkäsittelytieteen osa-alueen.

Kyseessä on poikkitieteellinen tutkimusalue, joka edellyttää erityisesti ihmisen käyttäytymisen, ajattelun ja ympäristönsä havainnoinnin periaatteiden tuntemista. Tässä esityksessä on mahdollista esittää näistä vain joitakin tärkeimpiä piirteitä.

4.3. Käyttöliittymän merkitys

Alkuaikojen tietokoneiden reikäkorteista ja -nauhoista, puhumattakaan kytkinpaneeleista ja binäärilukuja näyttävistä lamppuriveistä on käyty pitkä tie nyky-

[1] engl. human-computer interaction, HCI

päivän graafisiin käyttöliittymiin ja hiirellä osoittamiseen. Tietokoneita käyttävät työssään ihmiset, jotka eivät ole kiinnostuneita itse tietokoneen tai sen ohjelmiston sisäisestä toiminnasta, vaan haluavat päästä mahdollisimman helposti käyttämään sitä ilman erityistä koulutusta tai perehdyttämistä. Tämä asettaa suuret vaatimukset ohjelmien käyttöliittymien ja ohjeistojen suunnittelijoille.

Voidaan sanoa, että nykyisin hyvin suuri osa tietokoneiden käyttäjistä käyttää tietokoneita ainoastaan siksi, että heidän on pakko, jotta saisivat työnsä tehdyksi. Kun he eivät erityisesti nauti itse tietokoneen käytöstä, vaan se on heille työkalu muiden joukossa, sen käyttö ei saisi ainakaan tehdä työskentelyä vaikeammaksi. Tietokoneen tehtävä on siis auttaa käyttäjäänsä selviytymään työstänsä paremmin ja tuottavammin ja käyttöliittymän tehtävä on auttaa käyttäjää selviytymään tietokoneen käytöstä helpommin.

Hyvä käyttöliittymä ei synny ohjelmaan itsestään. Monet ohjelmistoja suunnittelevat henkilöt ovat itse tottuneet käyttämään hyvinkin monimutkaisia käyttöjärjestelmiä ja hyväksymään niiden kömpelyyden, koska tuntevat hyvin niiden taustan. Toisaalta he ovat luonnostaan, taipumustensa ja koulutuksensa pohjalta, myös erityisen kiinnostuneita ja hyvin perillä tietokoneiden ja ohjelmistojen toiminnasta ja käytöstä. Nämä seikat tulevat erityisen korostetusti esiin korkeakoulumaailmassa, jossa yleisesti käytetään UNIX-käyttöjärjestelmää lukuisine komentoineen ja komentorivipohjaisine apuohjelmineen.

Edellä mainituista syistä käy usein niin, että hyvin tarkkaan perustutkimukseen ja huolelliseen, useita henkilötyövuosia vaatineeseen ohjelmointiin perustuvat, vieläpä kattavan ja monipuolisen toimivuustestauksen läpikäyneet ohjelmat jäävät vähäiselle käytölle, kun niiden käyttöliittymät ovat vaikeita omaksua.

Hyvin suunnitellulla, nykypäivän tietokonetekniikkaa hyödyntävällä graafisella käyttöliittymällä ja siihen upotetulla ohjeistuksella ohjelmaan voidaan sisällyttää erittäin paljon toimintoja, jotka käyttäjä silti pystyy omaksumaan melko vaivattomasti, usein jopa pelkästään itsenäisesti tutustumalla. Ohjelman *käytettävyys* paranee selvästi.

4.4. Inhimilliset tekijät

Tässä kappaleessa tarkastellaan joitakin ihmisen perusominaisuuksia ja käyttäytymiseen liittyviä käsitteitä ja sitä, miten ne on huomioitava käyttöliittymää suunniteltaessa.

4.4.1. Kognitiiviset tekijät

Ihmismielessä tapahtuvan tietojenkäsittelytoiminnan eli tajunnan eri tekijöistä puhuttaessa käytetään psykologiassa lyhyesti termiä *kognitio* ja vastaavaa tieteenhaaraa kutsutaan *kognitiiviseksi psykologiaksi*. Termi sisältää laaja-alaisesti ihmisen kyvyn tehdä johtopäätöksiä ja ratkaista ongelmia sekä muokata ja säi-

lyttää tietoa siten, että se on myöhemmin helposti saatavilla ja hyödynnettävissä. Tavallaan kyse on siis oppimisen perusteiden tutkimisesta.

Graafiset käyttöliittymät ovat siinä mielessä kognitiivisia työkaluja, että ne pyrkivät tukemaan ja hyödyntämään ihmisen kognitiivisia taitoja. Siten niiden tärkeä ominaisuus on käytön luontevuus, jolloin itse työkalu muuttuu käytännössä huomaamattomaksi ja ihminen tuntee käyttävänsä käsiteltävää tietoa tai materiaalia suoraan. Tähän päästään vain silloin, kun ohjelmiston toiminta liittyy saumattomasti yhteen niiden prosessien ja rakenteiden kanssa, jotka osallistuvat tiedon käsittelyyn ihmismielessä.

4.4.2. Muisti

Kognitiivinen psykologia jakaa ihmisen muistin kolmeen päätyyppiin:

- *Sensoriset puskurimuistit* liittyvät aisteihin. Ne toimivat suurikapasiteettisina, mutta lyhytaikaisina puskureina tallettaen kaiken vastaanotetun informaation. Niillä ei juuri ole merkitystä tietojenkäsittelyn kannalta.

- *Työmuisti* on ihmisen tietojenkäsittelyn kannalta tärkeä, mutta sekä kapasiteetiltaan että varsinkin kestoltaan rajallinen. Käyttöliittymän suunnitteluun vaikuttaa tämän lyhytaikaisen muistin rajallisuus, sillä kerrallaan esimerkiksi näytöllä esitettävän informaation määrä muodostuu helposti liian suureksi.

- *Säilömuisti* tallettaa tietoa käytännöllisesti katsoen rajoittamattoman ajan, ts. pysyvästi. Käyttöliittymää suunniteltaessa on ratkaistava, millaiset seikat tukevat käyttörutiinien muistiin painamista ja mieleen palauttamista.

Komentorivipohjaisia ohjelmia tehtäessä käyttäjän muistin kapasiteetti yliarvioidaan helposti. Ohjelmaan sisällytetään kymmeniä erilaisia komentoja ja niihin kuhunkin vielä useita valitsimia, jolloin ohjelman kaikkien piirteiden käyttöön saamiseksi on muistettava liian paljon yksityiskohtaista tietoa. Suunnittelija ei itse huomaa tätä ongelmaa, koska hän ohjelmaa tehdessään omaksuu kaikki toiminnot vähän kerrallaan ja toisaalta tulee tehneeksi komennoista omien mieltymystensä mukaisia. Ryhmätyönä suunnitelluissa ohjelmissa ilmiötä saatetaan tosin kyetä lieventämään riittävällä vaatimusten määrittelyllä ja esisuunnittelulla.

Graafisen käyttöliittymän toiminnot voidaan helpommin suunnitella sellaiseksi, että niiden oppiminen ei juurikaan näytä kuormittavan käyttäjän muistia. Tämän katsotaan johtuvan siitä, että ne tukevat säilömuistin sisäisiä malleja, ns. *kognitiivisia skeemoja*. Skeemat toimivat ikään kuin talletusrunkoina tietoa tallennettaessa, jolloin skeemoja mukaileva informaatio tallentuu ja niihin sopimaton karsiutuu pois. Mieleen palautettaessa säilömuistissa olevasta pysyvästä tiedosta rakennetaan skeemojen avulla tuttuja rakenteita noudattelevia kokonaisuuksia. Siksi ihmiselle on helpompaa poimia oikeita komentoja valikoista kuin syöttää niitä täsmälleen oikeassa muodossaan komentoriville.

4.4.3. Huomiointi- ja vastaanottokyky

Ihminen havainnoi ympäristöään *aistiensa* avulla, joista tietokonetta käytettäessä kyseeseen tulevat näkö, kuulo ja nykyisin myös tuntoaisti. Seuraavassa keskitytään näköaistiin, johon graafiset käyttöliittymät pääasiassa perustuvat.

Havaitseminen riippuu kognitiosta, sillä ihminen muodostaa kuvan havaitsemastaan sekä uuden että aiemmin opitun informaation perusteella. Kuva 11 [Pre94] esittää tekstiä, jossa ei ole muuten mitään mieltä, mutta katsoja olettaa siinä kuitenkin olevan joitain oppimiansa sanoja. Lisäksi kuva on osoitus havaintojen tilanne- ja ympäristösidonnaisuudesta, sillä sama merkki tulkitaan kahdella eri tavalla riippuen ympäröivistä merkeistä.

Kuva 11: Havaintojen ympäristösidonnaisuus.

Kognitio vaikuttaa myös siihen, mihin katse näyttöruudulla helpoimmin kohdistuu. Koska länsimainen ihminen on tottunut lukemaan tekstiä vasemmalta oikealle ja ylhäältä alas, tämä on pyrittävä ottamaan huomioon sijoittamalla kohteet tärkeys- tai käyttöfrekvenssijärjestyksessä alkaen vasemmasta yläkulmasta.

Havaitsemiseen liittyvät läheisesti myös *tarkkaavaisuus* ja *reagointikyky*. Ihmisellä on erinomainen kyky keskittyä olennaiseen suuren informaatiotulvan joukosta, esimerkkinä vaikkapa tietyn keskustelun seuraaminen suuressa, hälisevässä ihmisjoukossa. Oppiminen parantaa vielä tuntuvasti tätä kykyä. Hyvin opittuja tehtäviä kutsutaan *automaattisiksi*, jolloin niiden suorittaminen sujuu, vaikka ajatukset jakautuisivat laajemmallekin. Ohjelmistojen käytön muuttuminen automaattiseksi on tavoiteltava ominaisuus ja edesauttaa jo aiemmin mainittua käyttöliittymän katoamista ja huomion suuntautumista sen sijaan lähes pelkästään itse tiedon käsittelyyn.

4.5. Graafisen käyttöliittymän suunnitteluperiaatteita

Käyttöliittymän – olipa se sitten graafinen tai komentorivipohjainen – suunnittelussa on tärkeää omaksua tietynlainen asenne, joka toimii kantavana johtoajatuksena läpi koko suunnitteluprosessin. Käyttäjää ei saa väheksyä ajattelemalla, että koska ohjelma on nerokkaasti tehty ja toimii luotettavasti, ei haittaa, vaikka se on vaikeasti käytettävä. Toisaalta ei myöskään riitä, että tehdään ohjelmalle kauniilta näyttävä graafinen käyttöliittymä, jonka avulla se on helpompi saada kaupaksi, mutta joka ei lopulta kuitenkaan tuo sen käytettävyyteen mitään lisäarvoa. Seuraavassa esitellään kolme periaatetta, jotka ovat hyviä noudatettavaksi graafisen käyttöliittymän suunnittelussa.

4.5.1. Käyttäjäkeskeinen suunnittelu

Käyttäjäkeskeisellä suunnittelulla tarkoitetaan sitä, että ensisijaisesti pyritään huomioimaan tulevat käyttäjät ja heidän psykofyysiset ominaisuutensa – vahvuudet ja heikkoudet – sekä tarpeensa. Tämä edellyttää, että tutustutaan niihin todellisiin ihmisiin, jotka ohjelmaa joutuvat käyttämään ja otetaan avoimesti vastaan kaikki kommentit. Mahdollisimman aikaisessa vaiheessa on pyrittävä saamaan käyttökokemuksia prototyyppien avulla ja mieluiten todellisessa käyttöympäristössä todellisella datalla, jolloin myös häiriötekijät tulevat otetuksi huomioon.

Käyttäjäkeskeisyyteen sisältyy olennaisena osana ohjelman *käytön ohjeistus*, josta kerrotaan tarkemmin kappaleessa 4.6.

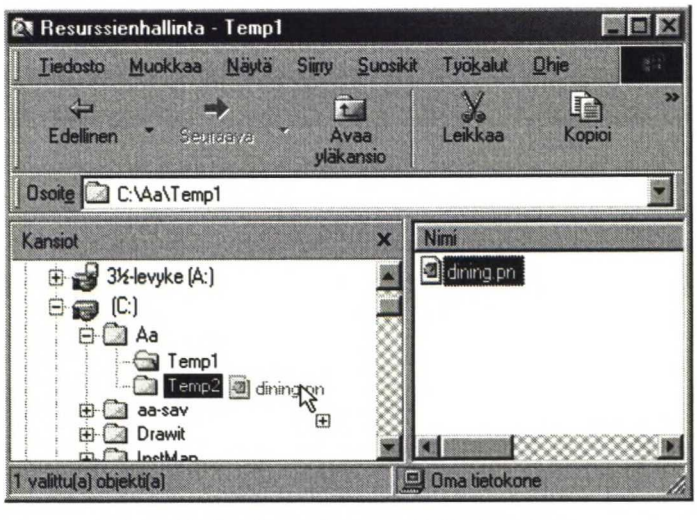
4.5.2. Tietokeskeinen toiminta

Edellä on jo viitattu siihen, että käyttöliittymän avulla käyttäjä voidaan saada tuntemaan, että hän ikään kuin käsittelee taustalla olevaa tietoa suoraan, ilman välissä olevaa tietokonetta ja ohjelmaa.

Tietokeskeisyyden eroa perinteiseen komentokeskeisyyteen kuvaa seuraava esimerkki, jossa tiedosto kopioidaan hakemistosta toiseen PC-koneessa. Komentorivipohjaisessa MS-DOS-käyttöjärjestelmässä se tapahtuu komennolla

```
copy c:\Aa\Temp1\dining.pn c:\Aa\Temp2\
```

eli ensin kerrotaan *mitä* tehdään, sitten vasta *mille* se tehdään. Graafisessa Windows-järjestelmässä sen sijaan toimintafilosofia on erilainen:



Kuva 12: Tiedoston kopiointi Windowsissa.

Ensin osoitetaan tiedostoa `dining.pn`, minkä jälkeen kopiointi voidaan suorittaa hiiren avulla vetämällä. Kopioinnin sijasta voitaisiin myös valita jokin

muu toimenpide. Oleellista on siis se, että *tieto* valitaan ensin, sitten vasta tietoon kohdistuva *toiminto*.

4.5.3. Standardointi ja suositukset

Vaikka graafisia käyttöliittymiä on ollut olemassa jo yli kymmenen vuotta, alan standardointi ei ole vakiintunut. Ainoa kansainvälinen varteenotettava standardi on ISO:n[1] vuonna 1996 julkaisema näyttöpäätteiden ergonomiaa ja käytettävyyttä käsittelevä standardi [ISO96]. Sen sijaan aiheesta on olemassa runsaasti kirjallisuutta ja paikallisia suosituksia, esimerkiksi TKK:n käytettävyyslaboratorion kirjoittama [URG00], joka on oppimateriaalina käyttöliittymien suunnittelua käsittelevillä tietotekniikan opintojaksoilla.

Alan selkeimmäksi teollisuusstandardiksi noussee eittämättä Microsoftin Windows-käyttöjärjestelmä, etenkin sen versio 95, jonka piirteitä on havaittavissa lähes kaikissa nykyisin käytössä olevissa graafisissa käyttöympäristöissä. Siinä toimivat ohjelmat noudattavat tiettyä samankaltaisuutta esimerkiksi valikoiden järjestyksessä, mikä on seurausta siitä, että ohjelmien tekoon on olemassa hyviä, hyvin dokumentoituja sovelluskehittimiä, esimerkiksi Visual Basic [Mic99], jotka ohjaavat tekemään käyttöliittymän tietyllä tavalla.

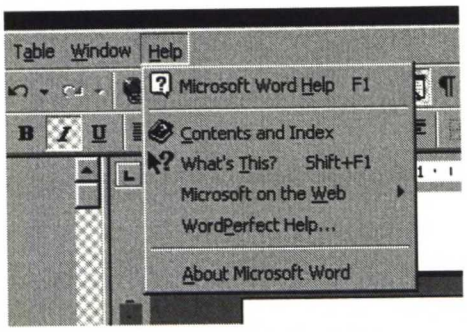
Vaikkakaan siis käyttöliittymän suunnittelua ei ohjaa mikään virallinen standardi, on kuitenkin syytä ottaa huomioon pitkän tuotekehittelyn ja käyttäjätutkimuksiin perustuvan palautteen muodostama vakiintunut käytäntö tehdä graafiset käyttöliittymät siten, että käyttäjä saa mahdollisimman suuren hyödyn jo omaksumastaan muiden ohjelmien käyttökokemuksesta.

4.6. Käyttöohjeet ja opasteet

Tietokeskeinen toiminta tekee ohjelmasta luonnostaan sellaisen, että sen käyttäminen enimmäkseen sujuu tarvitsematta tukeutua mihinkään ohjeisiin. Aloitteleva käyttäjä ja tottuneempikin silloin tällöin, varsinkin pitkän käyttötauon jälkeen, tarvitsee kuitenkin jonkinlaista apua alkuun päästäkseen.

Perinteisesti ohjelmista on kirjoitettu vaihtelevan laajuisia käyttöopaskirjoja ja erilaisia pikaohjekortteja tai –vihkosia. Näiden lukumäärä ja yleisyys on kuitenkin kaiken aikaa vähentynyt graafisten käyttöliittymien myötä ja nykyisin turvaudutaan lähes pelkästään ohjelmiin *sisäänrakennettuihin ohjeistoihin.* Seuraavassa kuvassa on esimerkkinä Microsoft Word tekstinkäsittelyohjelman ohjevalikko:

[1] International Organization for Standardization



Kuva 13: Word:in ohjevalikko.

Kuten esimerkistäkin näkyy, ohjevalikoihin on useimmiten sisällytetty vaihtelevan tasoisia ohjeistoja eritasoisia käyttäjiä ja erilaisia käyttötilanteita silmällä pitäen. Äärimmäisenä vaihtoehtona on yhteyden otto Internetin kautta valmistajan kotisivuilla oleviin ohjeistoihin, usein kysyttyjen kysymysten[1] luetteloihin ja jopa keskusteluryhmiin.

Kehittynein ohjetyyppi on ns. *tilannekohtainen ohje*[2], jonka voi aktivoida tietyllä pikavalintapainikkeella (usein F1) kesken jonkin tehtävän suorituksen ja siten saada yksityiskohtaista apua joutumatta selaamaan läpi pitkiä hakemistoketjuja.

[1] engl frequently asked questions, FAQ
[2] engl. context sensitive help

5. EMMA-kääntäjän muutostyö

Osana tätä työtä oli muuttaa olemassa oleva EMMA-kääntäjä sellaiseksi, että se generoi TNSDL-kielisestä lähdetiedostosta, tai paremminkin TNSDL-kääntäjän tuottamista FCF- ja TESTP-tiedostoista (ks. kappale 3.3) MARIA:lle kelpaavan Petri-verkkomallin.

Tässä luvussa kuvataan EMMA-kääntäjään tehtyjä muutoksia. Tarkastelu painottuu PROD- ja MARIA-kielten välisistä eroavaisuuksista johtuviin ohjelmointiteknisiin seikkoihin. Sitä, miten EMMA kääntää TNSDL:n rakenteita Petri-verkoksi ei kuvata lainkaan, koska siihen liittyvä osa ohjelmaa säilyi oleellisesti ennallaan. Näitä seikkoja on kuvattu mm. viitteissä [HOPV95], [Jyr97] ja [Mal97].

Mainittakoon, että samanaikaisesti tämän muutostyön kanssa oli tietojenkäsittelyteorian laboratoriossa tekeillä erillinen erikoistyö [Koi01], jossa EMMA:an tehtiin TNSDL:n version 4.0 vaatimat muutokset. Kyseisen työn raportissa kuvataan myös EMMA:n rakennetta hieman tarkemmin kuin tässä.

5.1. EMMA-kääntäjän rakenne

Käännös TNSDL-kuvauksesta Petri-verkkomalliksi tapahtuu karkeasti seuraavien vaiheiden mukaan:

1. TESTP-tiedoston tulkinta ja symbolitaulun generointi.
2. FCF-tiedoston tulkinta ja syntaksipuun generointi.
3. Optioiden tulkinta.
4. Petri-verkkomallin generointi syntaksipuusta.
5. Petri-verkon jälkikäsittely.
6. Petri-verkon tulostus.
7. Mallintarkastusominaisuuksien lisäys.

Vaiheet 1 - 3 voidaan ajatella eräänlaiseksi *esikäsittelyasteeksi*, jolloin pääosan varsinaisesta kääntäjästä muodostavat vaiheet 4 – 7. Kappaleessa 5.2 kuvatut PROD- ja MARIA-kielten väliset erot tulevat esiin lähes yksinomaan käännösvaiheessa lukuunottamatta joitakin pieniä esivaiheessa tehtyjä valmisteluita.

5.2. PROD- ja MARIA-kielten erot

Seuraavissa kappaleissa kuvataan ne erot PROD:in ja MARIA:n Petri-verkkokuvauskielten välillä, joista aiheutuvat muutokset EMMA-kääntäjään tehtiin tähän työhön sisältyen. MARIA:n mahdollistamien kehittyneiden tietotyyppien, kuten jonotyypin käyttöönotto jätettiin jatkokehityksen aiheeksi.

Tässä ei esitetä läheskään kattavasti mallinnuskielten eroja, vaan ainoastaan siinä laajuudessa, kuin on tarpeen EMMA:an tehtyjen muutosten ymmärtämiseksi.

5.2.1. Muuttujien yms. nimet

PROD-kielessä muuttujien nimeämissääntö on seuraava [VHHP95]:

> "Muuttujanimet alkavat kirjaimella, alleviivausmerkillä ("_") tai dollarimerkillä ("$") ja voivat jatkua halutulla määrällä kirjaimia, numeroita, alleviivausmerkkejä tai dollarimerkkejä. Isot ja pienet kirjaimet erotetaan toisistaan."

MARIA-kielessä [Mäk01] nimeämiskäytäntö on erittäin vapaa sallien esimerkiksi nimen `bool` edustaa tietotyyppiä, paikkaa, transitiota tai muuttujaa, samoin kuin sellaiset lainausmerkein rajoitetut nimet, jotka sisältävät jopa välilyöntejä tms. Tässä työssä päädyttiin kuitenkin noudattamaan seuraavaa yksinkertaista sääntökohtaa:

> "Mikä hyvänsä merkkijono muotoa `'[A-Za-z_][A-Za-z0-9_]*'`, joka ei ole varattu sana, on kelvollinen nimi."

Tämä sulki pois mahdollisuuden käyttää dollarimerkkejä nimissä, joten ne korvattiin alleviivausmerkeillä.

5.2.2. Vakioiden määrittely

PROD-kielessä voidaan käyttää C-kielen tapaa määritellä vakioita:

```
#define PIDMAX 10
#define TIMERMIN 11
```

mutta MARIA-kielessä `#define` on varattu suppeampaan käyttöön määrittelemään muuttujia käytettäväksi `#ifdef-#endif`-rakenteissa eli ohjaamaan ehdollista kääntämistä. MARIA-kielessä on kutakin vakiota kohden otettava käyttöön alustettu muuttuja, esim.

```
unsigned PIDMAX = 10;
unsigned TIMERMIN = 11;
```

Esimerkistä näkyy myös MARIA-kielen vaatima puolipiste jokaisen lauseen lopussa.

5.2.3. Monikko

Koska EMMA-kääntäjä on alun perin tehty PROD:ia varten, ainoa sen generoimissa malleissa käytetty tietotyyppi on yksi- tai useampialkioinen monikko. Tästä on poikettu muutetussa EMMA-kääntäjässä ainoastaan siten, että koko-

naislukualkioiden lisäksi on käytetty loogisia alkioita. PROD-kielessä monikko ilmaistaan esimerkiksi seuraavasti:

```
<. 1 .>
<. 1 , 2 .>
```

ja vastaavasti MARIA-kielessä

```
1
{ 1 , 2 }
```

ts. rajoittimina ovat aaltosulkeet, joita yksikössä ei käytetä lainkaan, koska sen korvaa yksinkertainen kokonaislukutyyppi. Kaksikko ja sitä leveämpi monikko on sen sijaan rakenteinen tietotyyppi.

5.2.4. Tietotyyppien määrittely

PROD-kielessä ainoa tietotyyppi on kokonaislukujen monikko. MARIA-kielessä tyyppivalikoima on suurempi.

MARIA-kielessä voidaan käyttää valmiiksi määriteltyjä tyyppejä `bool`, `int`, `unsigned` ja `char` ilman määrittelyä. Käytännössä on kuitenkin yleensä tarpeen määritellä lisäksi tietotyyppejä, jotka ovat kokonaislukujen suppeampia osajoukkoja, esimerkiksi

```
typedef unsigned (PIDMIN..PIDMAX) pid_range;
typedef unsigned (0..TIMERMAX) spid_range;
typedef unsigned (0..QUELEN) que_range;
typedef unsigned (SIGMIN..SIGMAX) sig_range;
typedef unsigned (0..VALUEMAX) value_range;
typedef unsigned (USEDPIDS+1..PIDMAX) pool_range;
```

Kaikki rakenteiset tietotyypit on ehdottomasti määriteltävä, esimerkiksi

```
typedef struct {
  pid_range x, spid_range y, que_range z,
  sig_range v, value_range w } que1_t;
```

määrittelee viisikon, jonka alkiot kuuluvat kukin eri kokonaislukujen osajoukkoon ja

```
typedef struct {
  pool_range x, bool y } pidp_t;
```

määrittelee tietueen, jonka ensimmäinen alkio kuuluu kokonaislukujen osajoukkoon ja toinen alkio on looginen muuttuja.

5.2.5. Paikan määrittely

PROD-kielessä paikka määritellään seuraavasti [VHHP95]:

```
#place nimi [lo(alaraja)] [hi(yläraja)] [mk(alkumerkintä)]
```

missä hakasulkeissa olevat osat kukin voidaan erikseen jättää pois. Esimerkiksi

```
#place Process$V$Self lo(<.PIDMIN,PIDMIN.>)
    hi(<.PIDMAX,PIDMAX.>) mk(<.1,1.> + <.2,2.>)
```

määrittelee paikan **Process$V$Self**, jossa merkit ovat kaksikkoja. Kaksikon kummankin alkion mahdollisten arvojen alaraja on vakio `PIDMIN` ja yläraja vakio `PIDMAX`. Alkumerkinnässä paikka sisältää kaksikot <.1,1.> ja <.2,2.>.

MARIA-kielessä paikka määritellään seuraavasti [Mäk01]:

```
place nimi rajoitus* tyyppi [const] [: merkintäluettelo]
```

missä hakasulkeissa olevat osat kukin voidaan erikseen jättää pois ja tähti tarkoittaa, että rajoituksia voi olla yksi, useita tai ei yhtään. *Tyyppi* voi olla aiemmin määritellyn tyypin nimi tai tyypin määrittely. Sana `const` ilmaisee tarvittaessa, että paikan sisältö on vakio. Edellinen esimerkki MARIA-kielellä on seuraavanlainen:

```
place Process_V_Self self_t : {1,1},{2,2};
```

missä tyyppi `self_t` on aiemmin määritelty:

```
typedef struct {
  pid_range x, pid_range y } self_t;
```

Esimerkistä käy ilmi myös erilainen tapa ilmaista monikoiden formaali summa: PROD:issa käytetään "+"-merkkiä ja MARIA:ssa pilkkua.

5.2.6. Transition määrittely

PROD-kielessä transitio määritellään seuraavasti [VHHP95]:

```
#trans nimi [fd(lista)]
  [in {paikan_nimi : lauseke ; ... }]
  [out {paikan_nimi : lauseke ; ... }]
  [gate ehtolauseke ;]
  [comp C-ohjelmalohko ]
#endtr
```

missä hakasulkeissa olevat osat kukin voidaan erikseen jättää pois. Comp-lohko voi sisältää mitä tahansa C-ohjelmalauseita. Esimerkki (mukailtu EMMA-mallista):

```
#trans Automaton$204$Trans0$1$Dec$10$1$Cond$ShadowElse$0
   in {
      V$193: <.pid,FromValue1.>;
      Process$129$Assign17: <.pid,state.>;
      RequireTheGlobalLock: <.pid,glockstate.>;
      SeenCritical: <.pid,sc.>;
      }
   out {
      V$193: <.pid,ToValue1.>;
      Automaton$204$Trans0$1$Dec$10$1: <.pid,state.>;
      RequireTheGlobalLock: <.pid,sc.>;
      SeenCritical: <.pid,sc.>;
      }
   gate (FromValue1!=0) && (FromValue1!=2);
   comp {
      if (FromValue1==3) ToValue1 = 7 else ToValue1 = 8;
      Accept();
      }
#endtr
```

MARIA-kielessä transitio voidaan määritellä tämän työn kannalta seuraavasti (viitteen [Mäk01] määritelmä on huomattavasti monipuolisempi):

```
trans nimi
  [in {paikan_nimi : lauseke ; ... }]
  [out {paikan_nimi : lauseke ; ... }]
  [gate ehtolauseke ] ;
```

missä hakasulkeissa olevat osat kukin voidaan erikseen jättää pois. On siis erityisesti huomattava, että PROD-kielen comp-lohkoa ei ole. Edellinen esimerkki MARIA-kielellä on seuraavanlainen:

```
trans Automaton_204_Trans0_1_Dec_10_1_Cond_ShadowElse_0
   in {
      V_193: {pid,FromValue1};
      Process_129_Assign17: {pid,state};
      RequireTheGlobalLock: {pid,glockstate};
      SeenCritical: {pid,sc};
      }
   out {
      V_193: {pid,(FromValue1==3)?7:8};
      Automaton_204_Trans0_1_Dec_10_1: {pid,state};
      RequireTheGlobalLock: {pid,sc};
      SeenCritical: {pid,sc};
      }
   gate (FromValue1!=0) && (FromValue1!=2);
```

Esimerkissä ollut comp-lohko on korvattu paikkaan **V_193** johtavalla kaarella olevan kaksikon toisessa alkiossa valintalausekkeella.

5.2.7. Monijoukon alkion kerroin

PROD-kielessä monijoukon alkion esiintymien lukumäärä lausekkeessa esitetään seuraavasti:

```
kerroin <. [a1 , a2 , ... ] .>
```

missä `kerroin` on kokonaislukuarvoinen lauseke.

MARIA-kielessä on vastaava esitys seuraavanlainen:

```
kerroin # merkintä
```

ts. #-merkki tarvitaan erottamaan kerroin merkinnästä, koska merkinnän ympärillä ei välttämättä ole mitään erottimia, kuten PROD-kielessä.

5.2.8. Mallintarkastus

PROD:issa on kaksi mahdollista tapaa suorittaa mallintarkastusta saavutettavuusgraafin generoinnin aikana [VHHP95]: `tester`-paikan käyttäminen ja verifiointikaavojen sisällyttäminen malliin. EMMA:ssa on käytetty `tester`-paikkaa

```
#place TheEMMAtester lo(<.0.>) hi(<.20.>) mk(<.0.>)
```

ja vastaavasti erilaisten erikoistilanteiden tarkastamiseksi määrittelyä

```
#tester TheEMMAtester reject(<.1..20.>)
```

mikä tarkoittaa, että saavutettavuusgraafin generointi lopetetaan, jos jossain transitiossa paikan **TheEMMAtester** sisällöksi asetetaan arvo välillä 1 – 20. Lukkiutumat tarkastetaan määrittelyllä

```
#tester TheEMMAtester deadlock(<.0.>)
```

Samat toiminnot saadaan MARIA-mallissa aikaiseksi määrittelyillä

```
reject !(0 subset (place TheEMMAtester)) && fatal;
```

ja

```
deadlock fatal;
```

5.2.9. Tyyppimuunnokset

PROD:issa kaikki monikkojen alkiot ovat etumerkittömiä kokonaislukuja ja siten aina samaa tyyppiä keskenään. MARIA sen sijaan on erittäin tarkka siitä,

että esimerkiksi vertailtavat muuttujat ovat keskenään samaa tyyppiä. Siksi joudutaan joskus tekemään tyypinmuunnos[1] seuraavasti:

```
is tyyppi muuttuja
```

Esimerkiksi `gate`-lauseessa

```
gate is npid_range outpid == is npid_range FromValue1
```

varmistetaan, että vertailu voidaan suorittaa ja muutetaan molemmat vertailtavat tyyppiin `npid_range`. Muutoin, riippuen mistä muuttujat ovat transitioon tulleet, niiden tyypit eivät ehkä olisi samat.

5.3. Muutokset EMMA-kääntäjään

Seuraavissa kappaleissa kuvataan lyhyesti edellä kuvatuista PROD- ja MARIA-kielten eroista johtuvat EMMA-kääntäjään tehdyt muutokset. Kussakin tapauksessa viitataan asianomaiseen kohtaan edellä, jossa kyseisiä eroavaisuuksia on tarkasteltu perusteellisemmin.

5.3.1. Yksinkertaiset syntaktiset muutokset

Kaikki paikkojen ja transitioiden nimissä esiintyneet dollarimerkit ("$") korvattiin kauttaaltaan alleviivausmerkeillä ("_") [5.2.1].

Vakiomäärittelyt korvattiin määrittelemällä samannimiset kokonaislukutyyppiset alustetut muuttujat [5.2.2]. Joitakin uusia nimettyjä vakioita lisättiin havainnollisuuden parantamiseksi. Probe-komentojonojen määrittelyt poistettiin tarpeettomina.

Kaikki merkinnöissä esiintyneet monikkosulkeet ("<. .>") korvattiin pääsääntöisesti aaltosulkeilla ("{ }"), mutta yksiköistä, jotka MARIA-mallissa korvattiin kokonaislukujen osajoukkotyyppiä olevilla muuttujilla ja vakioilla sulkeet jätettiin pois [5.2.3].

Monijoukon alkion ja kertoimen väliin lisättiin "#"-merkki asianomaisiin lausekkeisiin [5.2.7].

Mallintarkastukseen liittyvät määrittelyt korvattiin MARIA-kielen mukaisilla [5.2.8].

5.3.2. Tyyppimäärittelyt

EMMA:n generoimassa PROD-mallissa esiintyy useita erilaisia kokonaislukujen osajoukkoja, jotka rajoittavat paikoissa olevien monikkojen arvoalueita. Niiden määrittelyt on kuitenkin piilotettu paikkamäärittelyihin. Koska MARIA-mallissa haluttiin tässä vaiheessa säilyttää mahdollisimman samanlaiset tieto-

[1] engl. type casting

tyypit, samat arvoalueet määriteltiin myös siinä. Monikot määriteltiin tietueina, joiden kentät ovat joitakin edellä mainituista kokonaislukutyypeistä (5.2.4).

Tietotyyppien määrittely etukäteen oli välttämätöntä, jotta paikkamäärittelyt voitiin toteuttaa riittävän helposti ohjelmallisesti, mikä on esitetty kappaleessa 5.3.4.

5.3.3. Tyyppimuunnokset

Kappaleessa 5.2.9 esitettyjä, MARIA:n edellyttämiä tyyppimuunnoksia jouduttiin lisäämään useaan sellaiseen kohtaan, jossa esiintyi esimerkiksi eri arvoaluetyyppejä edustavia muuttujia tai muuttujia ja kokonaislukuvakioita. Tämä tehtiin pääasiassa siten, että aina, kun MARIA:lta saatiin huomautus tyyppien yhteensopimattomuudesta, kyseinen kohta muutettiin rutiininomaisesti.

5.3.4. Paikkojen määrittelyt

Kappaleessa 5.2.5 esitetyistä PROD:in ja MARIA:n paikkamäärittelyistä voidaan havaita tiettyä samankaltaisuutta, jota on käytetty hyväksi muutostyössä.

EMMA-kääntäjässä on olemassa metodit merkkijonon sijoittamiseksi PROD-paikan ala- ja ylärajamäärittelyiksi ja näitä käytettiin sijoittamaan MARIA-paikkaan vastaavasti rajoitus- ja tyyppimäärittelyt. Rajoituksia tosin ei yleensä tarvita. Alkumerkintää varten on myös oma metodi, jolle annetaan parametrina nyt MARIA:n syntaksin mukainen merkintä. Mainittuja metodeja jouduttiin muuttamaan vain vähän, samoin kuin lopullisen paikkamäärittelyn tulostuksen suorittavaa metodia.

5.3.5. Transitioiden määrittelyt

Kappaleessa 5.2.6 esitetyistä PROD:in ja MARIA:n transitiomäärittelyistä havaitaan, että ne ovat pääpiirteiltään samat, kun jo mainitut muut muutokset tehdään kukin omalta osaltaan. Selkein ero on `comp`-lohkojen puuttuminen, mistä aiheutuneet lisäykset jouduttiin tekemään transitioiden jälkikaarille.

5.3.6. Comp-lohkojen korvaaminen

Kappaleessa 5.2.6 on esitetty tyypillinen `comp`-lohkon käyttötapaus, jossa esikaarilla olevien muuttujien arvojen perusteella sijoitetaan jälkikaarten muuttujille arvoja. Kaikki EMMA:n generoimissa malleissa esiintyneet tapaukset ovatkin periaatteessa tällaisia, mutta eroja on `if`-lauseiden syvyydessä ja lausekkeiden monimutkaisuudessa.

Monimutkaisin, tosin MARIA-kielen joustavuuden ansiosta toteutukseltaan lopulta varsin yksinkertainen esimerkki `comp`-lohkon korvaamisesta on paikan

PidPool käsittely luotaessa uutta TNSDL-prosessia. Seuraavassa on esitetty tämän aikaansaava transitio, josta vain osa kaarista on otettu mukaan:

```
#trans state$223$Input144$$Trans4$2$Create$129
   in {
      PidPool: <.3,in3.> + <.4,in4.> + <.5,in5.> + <.6,in6.> +
               <.7,in7.> + <.8,in8.> + <.9,in9.> + <.10,in10.>;
      }
   out {
      PidPool: <.3,in3.> + <.4,in4.> + <.5,in5.> + <.6,in6.> +
               <.7,in7.> + <.8,in8.> + <.9,in9.> + <.10,in10.>;
      Process$SaveLatest: <.newpid,0.>;
      }
   comp {
      if (in3 == 0) { in3 = 1; newpid = 3; Accept(); }
      else if (in4 == 0) { in4 = 1; newpid = 4; Accept(); }
      else if (in5 == 0) { in5 = 1; newpid = 5; Accept(); }
      else if (in6 == 0) { in6 = 1; newpid = 6; Accept(); }
      else if (in7 == 0) { in7 = 1; newpid = 7; Accept(); }
      else if (in8 == 0) { in8 = 1; newpid = 8; Accept(); }
      else if (in9 == 0) { in9 = 1; newpid = 9; Accept(); }
      else if (in10 == 0 ){ in10 = 1; newpid = 10; Accept(); }
      }
#endtr
```

Toiminnan oleellinen ajatus on, että luotaessa uusi prosessi etsitään käytettävissä olevien `pid`:ien luettelosta *ensimmäinen* vapaa, annetaan se uudelle prosessille ja merkitään samalla varatuksi. Vastaava on toteutettu MARIA-kielellä seuraavasti:

```
trans state_223_Input144__Trans4_2_Create_129
   in {
      PidPool: {3,i3},{4,i4},{5,i5},{6,i6},{7,i7},
               {8,i8},{9,i9},{10,i10};
      }
   out {
      PidPool: (place PidPool) minus
               {!i3?3:(!i4?4:(!i5?5:(!i6?6:(!i7?7:
               (!i8?8:(!i9?9:(10)))))))),false} union
               {!i3?3:(!i4?4:(!i5?5:(!i6?6:(!i7?7:
               (!i8?8:(!i9?9:(10)))))))),true};
      Process_SaveLatest:
               (!i3?3:(!i4?4:(!i5?5:(!i6?6:(!i7?7:
               (!i8?8:(!i9?9:(10)))))))),0};
       }
   gate !i3 || !i4 || !i5 || !i6 || !i7 || !i8 || !i9 || !i10;
```

Toteutuksessa on käytetty hyväksi MARIA-kielen joukko-operaatioita `minus` ja `union`, joiden avulla paikasta **PidPool** on poistettu ensimmäinen vapaa merkki ja lisätty vastaava varattu merkki. Transitioon on lisätty `gate`-ehto, joka estää transition virittymisen, jos yhtään `pid`:iä ei ole vapaana. Lausekkeet on saatu selkeämmiksi, kun paikan **PidPool** merkin tyyppi on muutettu siten, että tietueen jälkimmäinen kenttä on tyyppiä `bool`.

5.4. Muutokset käyttäjän kannalta

Käyttäjälle nyt tehdyillä muutoksilla ei ole näkyvää merkitystä, vaan EMMA-kääntäjää käytetään edelleen samoin kuin kappaleessa 3.3 on esitetty. Tulostus-tiedoston, johon generoitu Petri-verkkomalli talletetaan, nimeksi on syytä valita sellainen, että se kelpaa XMARIA:lle, eli päätteeksi `.pn`.

Todettakoon vielä, että EMMA-tulostus ei ole enää käyttökelpoinen, koska PROD:ia ja sen `probe`-ohjelmaa ei käytetä. Generoitujen Petri-verkkomallien tarkasteluun suositellaan MARIA:n käyttöä XMARIA-käyttöliittymän avulla, jolloin EMMA:n tuottamia pitkiä paikkojen ja transitioiden nimiä ei tarvitse kirjoittaa komentoriville.

6. XMARIA-käyttöliittymän suunnittelu ja ohjelmointi

Edellä luvussa 4 tarkasteltuja ihmisen ja tietokoneen välisen vuorovaikutuksen suunnitteluperiaatteita sovellettiin käytäntöön suunnittelemalla ja ohjelmoimalla MARIA-analysaattorille kokeellinen graafinen käyttöliittymä XMARIA. Tässä luvussa esitetään joitakin pääpiirteitä tehtävän ratkaisusta.

6.1. Suunnittelukriteerit

Tärkeimpinä kriteereinä luvussa 4 mainittujen lisäksi olivat käyttöliittymän pitäminen erillään taustalla olevasta MARIA-analysaattorista ja pyrkimys välttää tekemästä MARIA:an oleellisia muutoksia. Näin käyttöliittymän ylläpitoon voitaisiin käyttää sellaisia henkilöitä, jotka eivät ole perehtyneitä MARIA:n ohjelmointiin ja toisaalta MARIA:n jo entuudestaankin suurta ja monimutkaista ohjelmakoodia ei enää kasvatettaisi.

Mahdollisimman suuren käyttäjäkunnan saavuttamiseksi tärkeänä kriteerinä oli pyrkimys mahdollisimman täydelliseen laiteriippumattomuuteen ja siten mahdollisuuteen levittää käyttöliittymä saataville kaikkialla, missä MARIA:akin on mahdollista käyttää, pääasiassa UNIX- ja LINUX-ympäristöissä ja toissijaisesti, mikäli mahdollista, myös Microsoft Windows-ympäristössä.

Ylläpitoa ajatellen ohjelmasta haluttiin vielä tehdä mahdollisimman modulaarinen ja siten havainnollinen.

6.2. Toteutusvaihtoehtojen valinta

Luonteva valinta kaikki edellä luetellut suunnittelukriteerit täyttäväksi ratkaisuvaihtoehdoksi oli Java-kieli. Javaa on jo useiden vuosien ajan opetettu ensimmäisenä ohjelmointikielenä korkeakouluopiskelijoille, joten potentiaalinen ylläpitohenkilöstö kasvaa koko ajan voimakkaasti. Lisäksi Java on nykyaikainen olio-ohjelmointikieli, jolla kyetään nopeasti toteuttamaan rakenteeltaan selkeitä ohjelmia. Vastaava, jo pitempään käytössä ollut kieli on C++, jolla esimerkiksi EMMA ja MARIA pääosin ovat toteutetut, mutta sillä tehdyt ohjelmat ovat paljon vaikeammin omaksuttavia tulevaisuuden ylläpitohenkilöiden kannalta.

Javan valinnan C++:n sijasta ratkaisi kuitenkin ennen kaikkea se, että Javaan on olemassa suhteellisen helppokäyttöinen ja hyvin dokumentoitu AWT[1]-paketti, jolla yksinkertainen ikkunoitu käyttöliittymä voidaan toteuttaa selvästi helpommin kuin C++:lla. AWT on nykyisissä Javan versioissa syrjäytymässä uuden Swing-paketin tieltä, mutta se valittiin, jotta yhteensopivuus vanhempien Java-versioiden kanssa olisi turvattu.

[1] abstract window toolkit

Käyttöliittymän ohjeistus päätettiin toteuttaa Internet-sivustona, jolloin sitä voidaan selata myös käyttämättä itse ohjelmaa. Sivut toteutettiin HTML-kielellä[1] ohjelmoiden ja käyttäen mahdollisimman yksinkertaisia rakenteita, jolloin ne latautuvat nopeasti katseltavaksi ja ylläpito on helppoa ilman erityisiä editointityökaluja.

6.3. Java-ohjelmointikieli

Java syntyi Sun Microsystems-yhtiössä 1990-luvun alkupuolella kehitetyn, sulautettujen järjestelmien ohjelmointiin tarkoitetun Oak-kielen pohjalta ja sen määrittely [GJS96] ilmestyi vuonna 1996. Erääksi tärkeäksi sovellusalueeksi Java-kielelle katsottiin alusta alkaen – ja on sittemmin muodostunutkin – Internet-sivuille tehtyjen ohjelmoitujen toimintojen (sovelmat eli appletit) laadintamahdollisuus. Ehkä siitä syystä Javasta tuli hyvin pian erittäin suosittu ohjelmointikieli Internetin räjähdysmäisen kasvun myötä. Java-ohjelmoinnista on julkaistu suuri määrä kirjallisuutta, joista tätä työtä tehtäessä on käytetty viitteitä [Fla99], [HC97], [HC98], [PM00] ja [WIK99].

Java muistuttaa hyvin paljon C-kieltä, joten C:llä ohjelmoineet henkilöt omaksuvat sen käytön nopeasti. Java on olio-ohjelmointikieli, kuten myös C-kieleen pohjautuva C++, mutta Java ei muistuta mitenkään selkeästi C++:aa. On sanottu, että Java vastaa suunnilleen sitä, mitä C++:n olisi pitänyt C-kielen olio-ohjelmointiin tarkoitettuna laajennuksena alunperin olla, mikä toisin sanoen tarkoittaa käytännössä, että Java on paljon helppokäyttöisempi kuin C++.

6.3.1. Ohjelmointi Java-kielellä

Java on aidosti olio-ohjelmointikieli, mikä näkyy ensimmäiseksi siitä, että kaikki muuttujien määrittelyt ja suoritettava ohjelmakoodi on sijoitettava *luokkien* sisään. Luokissa olevat muuttujat eli *kentät* ja aliohjelmat eli *metodit* ovat aina kyseiseen luokkaan kuuluvia ja voivat olla luokan ulkopuolella näkyviä tai näkymättömiä. Kullakin luokalla on vähintään yksi erikoismetodi, *konstruktori*, jolla luodaan kyseisen luokan *ilmentymiä*.

Luokat voivat *periytyä* toisiltaan, jolloin ne saavat käyttöön perimänsä kantaluokan ominaisuuksia. Itse asiassa kaikki luokat, niin Javassa valmiina olevat kuin ohjelmoijan itsensä tekemätkin, periytyvät luokasta Object, mitä ei tarvitse erikseen ohjelmassa ilmoittaa. Lisäksi varsinkin Javan valmiiden kirjastoominaisuuksien tarvitsemia määrittelyitä on sijoitettu ns. *rajapintaluokkiin*, jotka sisältävät ainoastaan metodien oikeanmuotoiset määrittelylauseet argumentteineen, muttei lainkaan suoritettavaa ohjelmakoodia. Itse metodit on toteutettava perivässä luokassa.

Java on ohjelmoijan kannalta turvallinen kieli, mikä tarkoittaa käytännössä, että

[1] hypertext markup language

monilta esimerkiksi C++-ohjelmoinnissa esiintyviltä tyypillisiltä virheiltä vältytään. Osoittimia ei ole ja muistin käytön hallinta on automaattista. Luokkien ilmentymiä luotaessa tarvittava muistitila varataan järjestelmän toimesta ja erityinen *roskienkeruujärjestelmä* huolehtii tilan vapautuksesta, kun sitä ei enää tarvita. Dynaamisia tietorakenteita varten on olemassa käyttökelpoisia valmiita kirjastoluokkia. Java-kääntäjä myös edellyttää tietyissä tapauksissa, kuten tiedostojen käsittelyssä, erityisten *poikkeuskäsittelijöiden* olemassaoloa ohjelmassa, joiden toiminta tosin on ohjelmoijan vastuulla.

Javan valmiit luokkakirjastot muodostavat yhdessä *sovellusrajapinnan* eli *Java API:n*[1] [API-W]. Erilaisia työkalulajeja on koottu pakkauksiksi, esimerkiksi `java.io`, joka sisältää tiedostojen ym. syöttö- ja tulostusvirtojen käsittelyä, `java.lang`, joka sisältää eräitä tietotyyppejä, kuten merkkijonot sekä jäljempänä kappaleessa 6.4.4 mainitut Runtime- ja Process-luokat ja `java.util`, joka sisältää mm. dynaamisia tietorakenteita.

6.3.2. Ohjelman kääntäminen ja suorittaminen

Java-kääntäjä generoi kustakin ohjelmoidusta luokasta ns. *tavukoodi*-tiedoston, jonka nimen pääte on yleensä `.class`. Tavukoodi on määritelty siten, että sen avulla pyritään takaamaan laiteriippumattomuus, ts. luokkatiedoston pitäisi olla suoritettavissa missä hyvänsä tietokoneessa, jossa on Java-tulkki eli *Java-virtuaalikone, Java-VM.*

Ohjelma suoritetaan käynnistämällä Java-tulkki argumenttina sen luokan nimi, joka sisältää `main`-metodin. Vain yksi tällainen metodi saa esiintyä ohjelmassa. Pääohjelmaluokan ja sen kutsumien muiden luokkien on löydyttävä tunnetun polun takaa, joka on joko nykyinen hakemisto tai ympäristömuuttujan `CLASSPATH` arvo.

6.4. XMARIA-käyttöliittymän ohjelmointi Java:lla

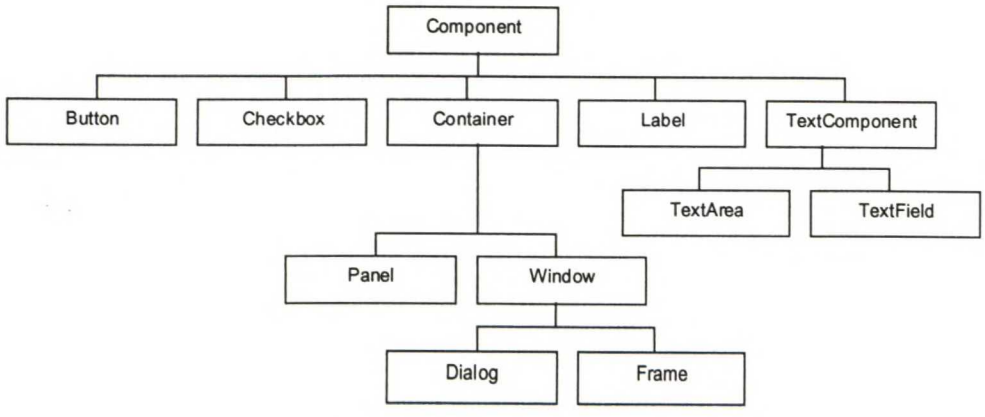
Javan alkuperäinen ja vieläkin laajalti käytössä oleva työkalu graafisten käyttöliittymien tekemiseen on AWT-luokkakirjasto, joka on suunniteltu toimimaan missä tahansa Java-ohjelmia suorittavassa tietokonejärjestelmässä. AWT tarjoaa peruselementit varsin monipuolisen käyttöliittymän aikaansaamiseksi, mutta koska sen on oltava laitteisto- ja järjestelmäriippumaton, on useissa kohdin jouduttu tyytymään varsin vaatimattomiin toimintoihin verrattuna useimpiin vastaaviin järjestelmäkohtaisiin käyttöliittymiin.

6.4.1. Peruselementit

Graafisessa käyttöliittymässä peruselementtinä on *komponentti* (luokka Component), jonka ominaisuuksiin kuuluu mm., että se voi reagoida erilaisiin käyt-

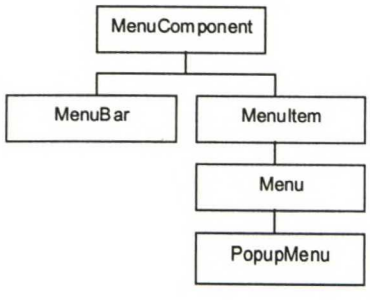
[1] application programming interface

täjän aiheuttamiin *tapahtumiin*, kuten hiiren näppäimen painallus. Useimmat AWT:n elementit ovat periytyneet Component-luokasta seuraavan kaavion mukaisesti:



Kuva 14: Osa AWT:n luokkahierarkiasta.

Erilaiset valikkoelementit, jotka ovat ikkunoidussa käyttöliittymässä oleellisia, eivät sisälly yllä esitettyyn luokkahierarkiaan, vaan periytyvät omasta MenuComponent-luokasta, kuten seuraavassa kaaviossa on esitetty:



Kuva 15: Osa valikko-elementtien luokkahierarkiasta.

6.4.2. Tapahtumien käsittely

AWT:lla tehdyissä ikkunoidussa käyttöliittymässä kaikki toiminnot käynnistyvät jostain käyttäjän aiheuttamasta tapahtumasta. Jokaiseen tapahtumaan liittyy sen *lähde*, esimerkiksi hiiren painikkeen painallus ja *kuuntelija*, rajapintaluokka, jossa määritelty, ohjelmoijan laatima metodi saa argumenttina tapahtumaluokan olion ja toteuttaa vaadittavan toiminnon.

Seuraavassa taulukossa on esitetty XMARIA:ssa käytettyjä tapahtumia:

Lähde	Tapahtumaluokka	Kuuntelija	Metodit
valikkovalinta	ActionEvent	ActionListener	actionPerformed
painikkeen aktivointi	ActionEvent	ActionListener	actionPerformed
näppäimistön painike	KeyEvent	KeyListener	keyTyped
hiiren painike	MouseEvent	MouseListener	mouseClicked
ikkunan muutos	WindowEvent	WindowListener	windowClosed

6.4.3. XMARIA:n tärkeimmät luokat

Pääohjelmaluokka Xmaria periytyy luokasta Frame ja sille kuuluvat käyttöliittymän pääikkuna valikoineen ja kaikki ponnahdusvalikot riippumatta siitä, missä ikkunassa ne avautuvat. Suurin osa tapahtumista aktivoidaan em. valikoista ja ne on koottu luokkaan MenuActions, joka toteuttaa rajapintaluokan ActionListener.

6.4.4. MARIA:n aktivointi ja kommunikaatio

XMARIA muodostaa taustalla olevan, komentorivipohjaisella käyttöliittymällä toimivan MARIA:n päälle eräänlaisen kuoren, jonka läpi käyttäjä kommunikoi MARIA:n kanssa.

MARIA aktivoidaan tarvittaessa Javan Runtime-luokan metodilla `exec`, joka palauttaa Process-luokan olion. Tämä toimii sen jälkeen tartuntapintana MARIA:n.

Kommunikaatiota varten MARIA:n syöttö ja tulostus, jotka kulkevat normaalisti `stdin`- ja `stderr`-virtojen kautta, on ohjattava XMARIA:lle, mikä tehdään Process-luokan metodeilla `getOutputStream` ja `getErrorStream`.

Pysäyttäminen tapahtuu lähettämällä MARIA:lle `exit`-komento tai Process-luokan metodilla `destroy`.

6.5. Muutokset MARIA:an

Ollessaan interaktiivisessa tilassa MARIA tulostaa *kehotteen*, joka on joko nykyisen tilan numero, jonka edellä on @-merkki ja jäljessä $-merkki tai joissakin tilanteissa pelkkä $-merkki. Kehotteen jäljessä ei ole rivinvaihtoa, koska normaalikäytössä on tarkoitus, että käyttäjä kirjoittaa kehotteen perään komennon. Tämä aiheuttaa ainakin LINUX-järjestelmässä sen, että XMARIA ei saa kehotetta sitä odottaessaan. Tämä on korjattu muuttamalla MARIA:a niin, että kehotteenkin jälkeen tulostuu rivinvaihto. Samalla lisättiin kehotteen alkuun alleviivausmerkki, jolloin XMARIA tietää aina varmuudella, että kyseessä on kehote, eikä esimerkiksi jokin muu @-merkillä alkava rivi.

Käyttöliittymän ohjelmisto saatiin huomattavasti yksinkertaisemmaksi tekemällä MARIA:an lisäksi muutos dump-komennon tulostukseen siten, että transition jälkeen tulostuu aina yksi ylimääräinen puolipiste omalle rivilleen.

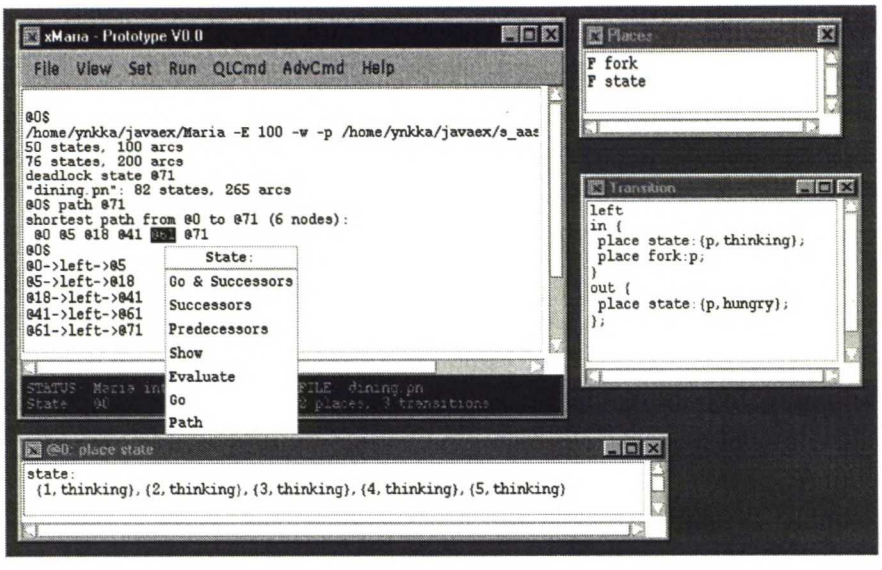
7. XMARIA-käyttöliittymän ominaisuudet

Tässä luvussa kuvataan laaditun kokeellisen XMARIA-käyttöliittymän ominaisuuksia ja käyttöä siinä laajuudessa kuin on tarpeen sen osoittamiseksi, miten MARIA:n käytettävyys paranee. Kyseessä ei siis ole varsinainen käyttäjän käsikirja, vaikka toimintojen esitysjärjestys noudattelee pääpiirteittäin yksinkertaisen mallin käsittelyskenaariota.

Esimerkkimallina käytetään pääasiassa kappaleessa 2.1.3 esitettyä aterioivien filosofien ongelman Petri-verkkomallia (Kuva 7 s. 11) esitettynä MARIA-kielellä (Kuva 10 s. 19), kuitenkin ilman **counter**-paikkaa. Lisäksi esitellään joitakin käyttöliittymän piirteitä suuremmilla, lähempänä käytäntöä olevilla malleilla, jotka tuovat selkeämmin esiin kyseisen piirteen käyttökelpoisuuden.

Korostettakoon, että vaikka joissakin kohdin esitetään MARIA:n komentorivipohjaisessa käyttöliittymässä esiintyviä, käytännössä havaittuja vaikeuksia, tämän esityksen tarkoitus ei ole niiden korostaminen, vaan pyrkimyksenä on ainoastaan herättää ideoita vastaavien piirteiden käyttäjäystävälliseen suunnitteluun tulevaisuudessa. Käyttötilanteita esittävät kuvat on otettu kuvaruudulta tilanteista, joissa XMARIA toimii eräässä tietojenkäsittelyteorian laboratorion Debian-LINUX-koneessa. Kyseiseen koneeseen on otettu SSH-yhteys[1] Windows98-käyttöjärjestelmällä ja X-WinPro-ohjelmistolla varustetulla PC-tietokoneella. Ikkunoiden ulkoasu vaihtelee käytännössä ohjelmistoympäristöstä riippuen.

7.1. XMARIA:n työpöytä



Kuva 16: XMARIA:n työpöytä..

[1] secure shell

XMARIA-käyttöliittymän työpöytä koostuu seuraavista ikkunoista:

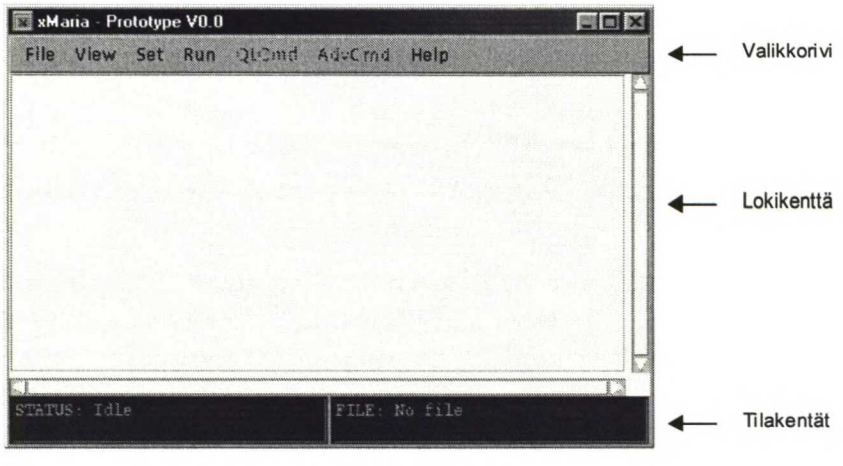
- Pääikkuna, jossa kommunikointi MARIA:n kanssa pääasiassa tapahtuu.
- Apuikkuna, jossa näytetään vaihtelevaa tietoa, kuten merkintöjä. Otsikko vaihtelee esitettävän tiedon mukaan.
- Paikkaluetteloikkuna (Places).
- Transitioikkuna (Transition).

Työpöytäajattelun mukaisesti jokaisen esillä olevan ikkunan kokoa ja paikkaa näytöllä voidaan vapaasti muuttaa tai ikkunoita sulkea (lukuunottamatta pääikkunaa), jolloin kulloinkin tarvittavat tiedot saadaan parhaiten näkyville. Erityisesti mainittakoon, että pää-, apu- ja transitioikkunat voidaan yhdestä painikkeesta maksimoida täyteen kokoon[1].

Näytössä olevien ikkunoiden lomittelulla käyttäjä voi optimoida työskentelyään ja saada näkyville kullakin hetkellä oleellisen tiedon. Jos jokin ikkuna piiloutuu muiden taakse, se palautuu automaattisesti päällimmäiseksi, kun kyseistä tietoa tarvitaan näytettäväksi.

7.2. Pääikkuna

Käyttöliittymän perustana on pääikkuna, jonka kautta käyttäjä pääasiassa antaa komentoja MARIA-analysaattorille. Sen lisäksi avataan tarvittaessa erilaisia apuikkunoita. Ikkunoiden itsenäisiä alueita, joihin tulostetaan informaatiota kutsutaan tässä esityksessä *kentiksi*.



Kuva 17: XMARIA:n pääikkuna.

Pääikkunan yläosassa on *valikkorivi*, keskellä *lokikenttä* ja alaosassa kaksi *tilakenttää*, joista vasemmanpuoleinen (`STATUS:`) ilmaisee taustalla olevan MARIA:n tilan ja oikeanpuoleinen (`FILE:`) sisältää tietoa käsiteltävänä olevasta mallitiedostosta. Tilailmoitus `STATUS: Idle` tarkoittaa, että MARIA:a ei ole vielä lainkaan käynnistetty.

[1] Maksimointipainikkeen toiminta vaihtelee eri ikkunointijärjestelmissä.

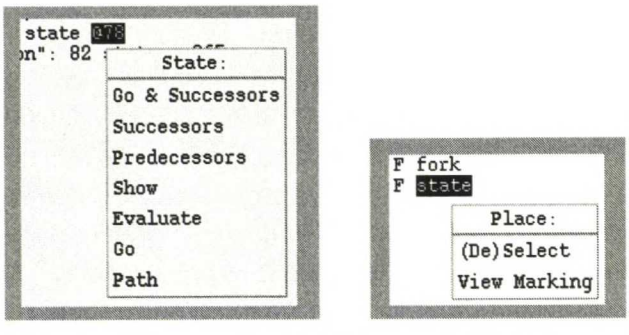
7.2.1. Lokikenttä

Lokikentästä käyttäjä näkee analyysin tulokset ja pystyy niiden pohjalta tekemään päätöksiä jatkotoimenpiteistä. Kenttään tulostuva teksti sisältää elementtejä, joita osoittamalla ja hiirellä aktivoimalla kyetään nopeasti antamaan komentoja. Reunoissa on vierityspalkit, joiden avulla voidaan tarkastella pitkiä rivejä ja palata takaisin tutkimaan aikaisempia tulostuksia.

Lähes kaikki kommunikaatio käyttöliittymän ja MARIA:n välillä tulostetaan lokikenttään. Tähän ratkaisuun on päädytty, koska MARIA:n tulostukset ovat varsin monimutkaisia ja osin vakiintumattomia ja toisaalta tällöin käyttäjä näkee mahdollisimman tarkasti, mikä analyysin tulos on. Jatkossa kuvataan joitakin tilanteita, joissa tulostusta kuitenkin muokataan tai jopa jätetään pois.

7.2.2. Tiedon osoittaminen, ponnahdusvalikot

XMARIA:n toiminnan perustana on *tietokeskeisyys* (ks. kappale 4.5.2), ts. kaikkien toimintojen ajatellaan kohdistuvan käsiteltävään tietoon, tässä tapauksessa Petri-verkkomalliin ja saavutettavuusgraafiin. Siksi on luontevaa, että toimintoja aktivoidaan osoittamalla käsiteltävää tietoa näyttöruudulla ja valitsemalla jokin kyseiseen tietoon kohdistuvista toimintomahdollisuuksista *ponnahdusvalikosta*.



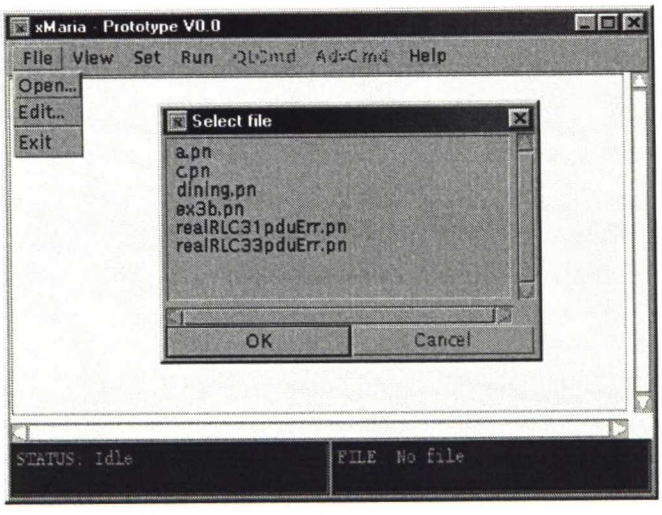
Kuva 18: TILA- ja PAIKKA-ponnahdusvalikot.

Yllä oleva kuva esittää TILA-ponnahdusvalikkoa, josta voidaan aktivoida siinä lueteltuja MARIA:n komentoja parametrina osoitettu tila ja paikkaluetteloikkunassa (kappale 7.7) toteutettua PAIKKA-ponnahdusvalikkoa, jonka avulla kyseisen paikan sisällyttämistä merkintöjen listaukseen voidaan ohjata tai paikan merkintä näyttää apuikkunassa.

Osoittamalla pääikkunan lokikentässä yksittäistä transitiota avataan transitioikkuna (kappale 7.8), jossa näytetään kyseisen transition Petri-verkkoesitys MARIA-kielellä.

7.3. Mallin avaaminen

Tietokeskeiseen ajattelutapaan sisältyy edelleen, ja sitä tukee se tosiasia, että kukin MARIA-istunto kohdistuu aina tiettyyn mallitiedostoon. Siksi hyvä vakiokäytäntö on ensimmäisenä toimenpiteenä käynnistyksen jälkeen kiinnittää istunnon aikana käsiteltävän mallitiedoston nimi, eli *avata* malli.



Kuva 19: Mallin avaaminen.

Avaamisen yhteydessä XMARIA käynnistää MARIA:n ja sen annetaan tarkistaa mallitiedoston syntaksi. Jos tiedosto on syntaktisesti virheetön, MARIA:lta saadaan luettelo kaikista siinä esiintyvistä paikoista ja transitioista. Tämän kommunikoinnin ajan vasemmassa tilaikkunassa on ilmoitus `STATUS: Loading symbols`.

Avaamisen jälkeen tiedoston nimeä ei tarvitse enää antaa, vaan kaikki toiminnot kohdistuvat automaattisesti siihen ja siitä generoituihin saavutettavuusgraafiym. tiedostoihin.

7.4. Virheitä mallissa, lähdetiedoston muuttaminen

Eräs MARIA:n käyttöä vaikeuttava piirre on se, että se jatkaa mallin tarkistamista loppuun saakka virheistä huolimatta, mikä aiheuttaa usein suuren määrän (jopa tuhansia) virheilmoituksia, vaikka mallissa olisi vain yksi virhe. Komentorivipohjaisessa käytössä ensimmäinen, koko tilanteen aiheuttanut virheilmoitus katoaa yleensä näyttöruudulta ja käyttäjä näkee vain joukon loppupään todennäköisesti aiheettomia, jopa virheellisiä ja siten harhaanjohtavia ilmoituksia.

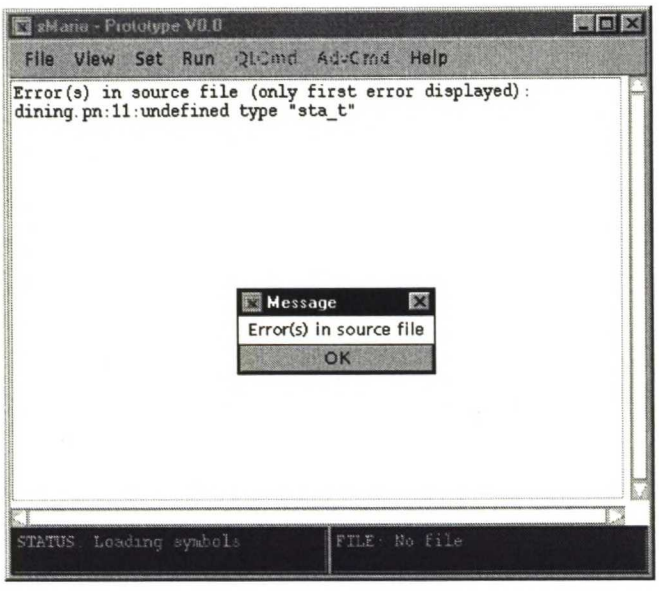
Jos esimerkiksi mallitiedoston rivillä 11 tehdään kirjoitusvirhe paikan **state** tyypin määrittelyssä:

```
place state (#phil_t) sta_t: phil_t p: { p, thinking };
```

ja yritetään generoida saavutettavuusgraafi, saadaan MARIA:lta seuraava tulostus:

```
jade ~ 54 % maria -v -w -b dining.pn -e exit
dining.pn:11:undefined type "sta_t"
dining.pn:11:cannot determine type of structured expression
dining.pn:11:undefined variable "thinking"
dining.pn:14:transition `left':undefined place "state"
dining.pn:14:transition `left':cannot determine type of structured expression
dining.pn:14:transition `left':cannot determine the type of variable "p"
dining.pn:14:transition `left':cannot determine the type of variable "thinking"
dining.pn:15:transition `left':undefined place "state"
dining.pn:15:transition `left':cannot determine type of structured expression
dining.pn:15:transition `left':cannot determine the type of variable "hungry"
dining.pn:18:transition `right':undefined place "state"
dining.pn:18:transition `right':cannot determine type of structured expression
dining.pn:18:transition `right':cannot determine the type of variable "p"
dining.pn:18:transition `right':cannot determine the type of variable "hungry"
dining.pn:19:transition `right':undefined place "state"
dining.pn:19:transition `right':cannot determine type of structured expression
dining.pn:19:transition `right':cannot determine the type of variable "eating"
dining.pn:19:transition `right':transition cannot be unified
dining.pn:22:transition `finish':undefined place "state"
dining.pn:22:transition `finish':cannot determine type of structured expression
dining.pn:22:transition `finish':cannot determine the type of variable "p"
dining.pn:22:transition `finish':cannot determine the type of variable "eating"
dining.pn:23:transition `finish':undefined place "state"
dining.pn:23:transition `finish':cannot determine type of structured expression
dining.pn:23:transition `finish':cannot determine the type of variable "p"
dining.pn:23:transition `finish':cannot determine the type of variable
dining.pn:23:transition `finish':"thinking"
dining.pn:26 errors
jade ~ 55 %
```

ts. MARIA luettelee kaikkiaan 26 virhettä, vaikka käyttäjä on selvästikin tehnyt vain yhden. Suurissa malleissa tämä ilmiö korostuu ja ylimääräisten virheiden määrä vaihtelee riippuen siitä, kuinka monessa kohdassa esim. määrittelemättä jäänyt muuttuja esiintyy. Käyttöliittymä tarjoaa ongelmaan erään ratkaisun:

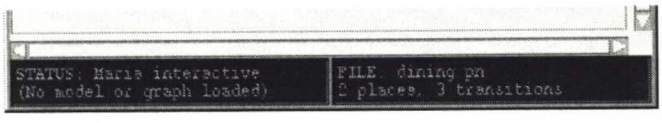


Kuva 20: Virhe mallissa.

Jos MARIA:lta saadaan ilmoitus virheestä, sen suoritus keskeytetään välittömästi ja loki-ikkunaan tulostetaan kyseinen virheilmoitus. Lisäksi käyttäjälle huomautetaan asiasta avaamalla viesti-ikkuna, joka on kuitattava ennen jatkoon pääsyä.

Käyttäjä voi korjata mallitiedoston virheitä tai muuten halutessaan myöhemminkin muuttaa tiedoston sisältöä valitsemalla File-valikosta Edit, jolloin käynnistyy *Emacs*-editori[1]. Editorista poistumisen jälkeen suoritetaan jälleen automaattisesti tiedoston avaus, ts. syntaksin tarkistus ja symbolien lataus.

7.5. Toiminta mallin avauksen jälkeen

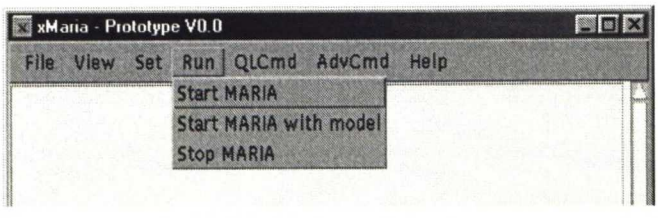


Kuva 21: Tilakentät mallin avauksen jälkeen.

Kun mallitiedosto on avattu virheettömästi, tilaikkunat osoittavat, että MARIA on taustalla odottamassa komentoja, mutta sen työtilaan ei ole vielä ladattu saavutettavuusgraafia tutkittavaksi. Avattuna on mallitiedosto `dining.pn`, jossa on 2 paikkaa ja 3 transitiota.

7.6. Saavutettavuusgraafin generointi

Mallitiedoston avattuaan käyttäjä voi valita useita erilaisia toimintavaihtoehtoja, joista pienten mallien tapauksessa tavallisin on saavutettavuusgraafin generointi. Suurille malleille ei yleensä kannata generoida koko graafia käyttöliittymän kautta, vaan parasta on käyttää ennalta generoitua graafia tai generoida vain osa sen tiloista.



Kuva 22: MARIA:n käynnistysvalikko.

Generointi aloitetaan valitsemalla Run-valikosta Start MARIA with model . Käynnistyskomento tulostuu lokikenttään ja tilaikkunassa näkyy `STATUS: Maria analysing`.

[1] Emacs on eräs yleisesti käytetty UNIX-järjestelmän tekstinmuokkausohjelma eli editori. Mikä hyvänsä muukin vastaava editori voidaan valita käynnistettäväksi XMARIA:sta.

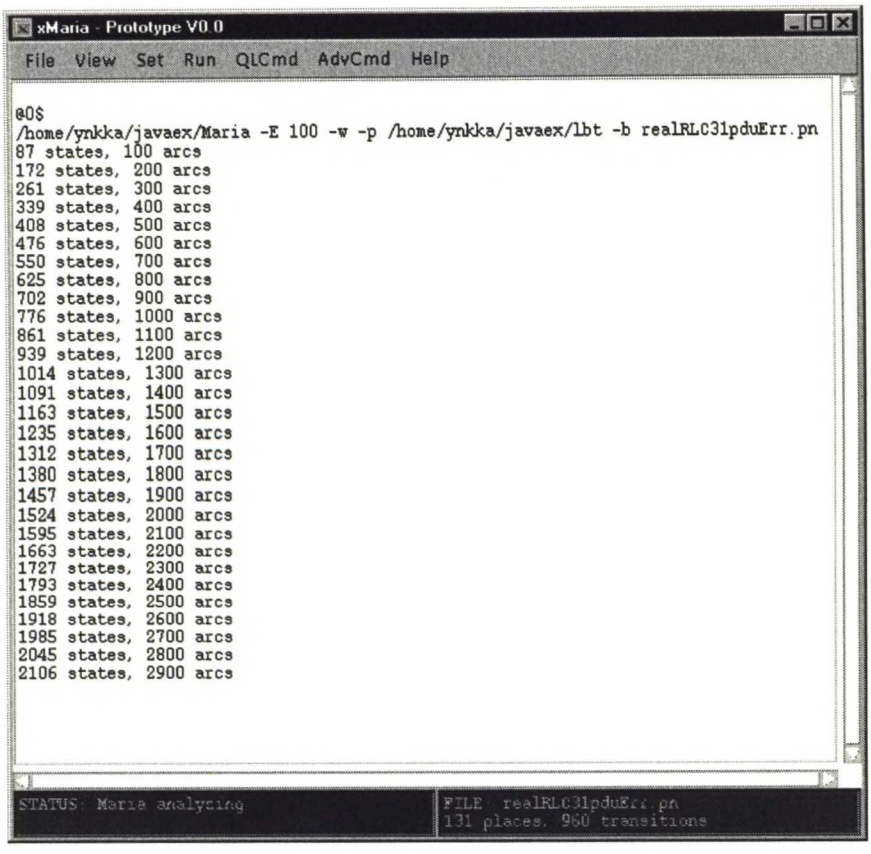
MARIA:n käynnistyskomennossa käytetään seuraavia komentoriviparametreja:

-E 100 edistymisestä ilmoitetaan aina saavutettavuusgraafiin lisättyjen sadan kaaren jälkeen

-w varoitukset jätetään tulostamatta

-p käytetään LTL-kaavamuunninta, jonka tiedostopolku on annettu

-b generointi suoritetaan breadth-first-menetelmällä

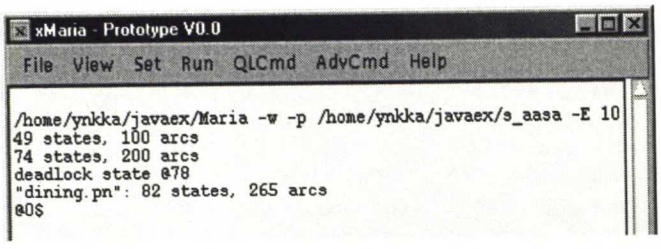


Kuva 23: Saavutettavuusgraafin generointi, suuri malli.

MARIA ilmoittaa saavutettavuusgraafin kasvusta loki-ikkunaan sadan generoidun kaaren välein, jolloin käyttäjä saa kuvan nopeudesta, jolla generointi edistyy.

Jos saavutettavuusgraafista alkaa tulla odotettua suurempi tai muutoin halutaan tarkastella vain graafin alkupään tiloja, generointi voidaan milloin tahansa keskeyttää valitsemalla Run-valikosta Stop MARIA , jolloin tilakenttään tulee ilmoitus STATUS: `Maria stopped`. Tällöin Maria on käynnistettävä uudelleen ja generoitu graafi ladattava tarkastelua varten.

Pienen mallin saavutettavuusgraafin voi yleensä generoida kokonaan:

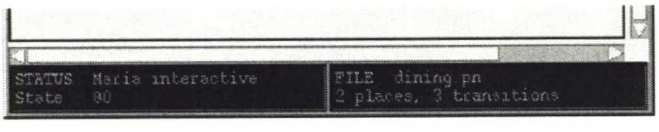


Kuva 24: Saavutettavuusgraafin generointi, pieni malli.

Tässä tapauksessa mallitiedostossa on rivi

```
deadlock true
```

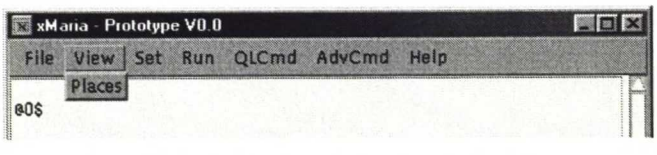
tarkoittaen, että pyydetään kaikkien mahdollisten lukkiutumien tarkastusta ja siten saavutettavuusgraafissa esiintyvä lukkiutuma tilassa `@78` on ilmoitettu. MARIA ilmoittaa myös aina onnistuneen prosessin jälkeen generoidun graafin tilojen ja kaarien lukumäärät.



Kuva 25: Saavutettavuusgraafi generoitu. MARIA odottaa komentoja.

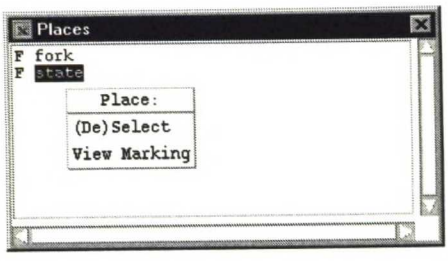
Generoituaan saavutettavuusgraafin MARIA jää interaktiiviseen toimintatilaan ja ilmoittaa tällä hetkellä tarkastelussa olevaksi graafin tilaksi, eli *nykytilaksi* tilan `@0`, joka vastaa mallinnetun Petri-verkon alkumerkintää. Nykytilan numero ilmaistaan aina myös vasemmanpuoleisessa tilaikkunassa.

7.7. Paikkaluetteloikkuna



Kuva 26: Paikkaluetteloikkunan avaaminen.

Valitsemalla View-valikosta Places avautuu apuikkuna, jossa luetellaan kaikki mallin paikat. Tämän apuikkunan ansiosta käyttäjän ei tarvitse missään vaiheessa kirjoittaa paikkojen nimiä. Ikkunasta on myös toinen versio, *paikanvalintaikkuna*, joka esitellään kappaleessa 7.13.



Kuva 27: Paikkaluetteloikkuna.

Kirjain F paikan nimen edessä tarkoittaa, että paikan merkintää ei näytetä Show-komennolla, kirjain T, että näytetään. Tämä edellyttää lisäksi, että kyseinen rajoitus on asetettu toimintaan (kappale 7.11). Valintoja voidaan muuttaa valitsemalla halutun paikan nimi ja avautuvasta ponnahdusvalikosta (De)Select . Vastaava toiminto tehdään MARIA:n komentokielessä komennolla

```
hide paikka1 [, paikka2 , ...]
```

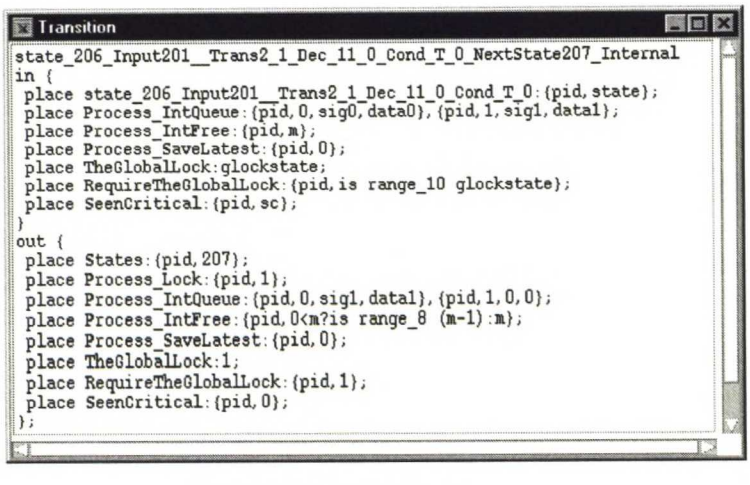
jokaista piilotettavaa paikkaa kohden tai vaihtoehtoisesti komennolla

```
hide; hide ! paikka1 [, paikka2 , ...]
```

ts. ensin piilotetaan kaikki paikat ja valitaan näytettävät paikat luettelemalla ne.

Näytettävät paikat voidaan valita etukäteen kirjoittamalla niiden nimet tiedostoon *<mallin nimi>*.ini, jonka sisältö luetaan samalla, kun malli avataan ja syntaksi tarkistetaan. Mallin käsittelyn aikaisia muutoksia ei nykyisellään talleteta kyseiseen tiedostoon, vaan käyttäjän on itse muutettava sen sisältöä.

7.8. Transitioikkuna



Kuva 28: Transitioikkuna.

Pääikkunan lokikentässä esiintyvää transition nimeä osoittamalla ja napsauttamalla hiiren painiketta saadaan näkyviin transtioikkuna, jossa on kyseisen tran-

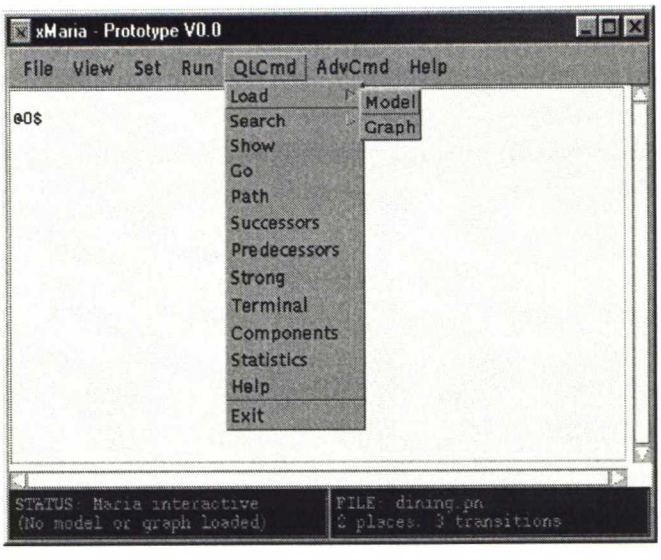
sition MARIA-kielinen esitys siinä muodossa, jossa se on saatu MARIA:lta mallin avauksen ja syntaksin tarkistuksen yhteydessä (dump-komennolla).

7.9. MARIA:n komennot

Edellä kappaleessa 7.2.2 esiteltiin TILA-ponnahdusvalikko, josta voidaan aktivoida eräitä MARIA:n komentoja parametrina osoitettu tila.

Valitsemalla valikkoriviltä QLCmd (Query Language Commands) saadaan valikko tärkeimmistä MARIA:n komennoista. Komennon aktivointi tämän valikon kautta antaa mahdollisuuden käyttää parametrina myös lokikentässä näkymättömiä tiloja.

Load Model- ja Load Graph-komennot eivät vaadi parametreja, vaan käyttävät avatun mallitiedoston nimeä.

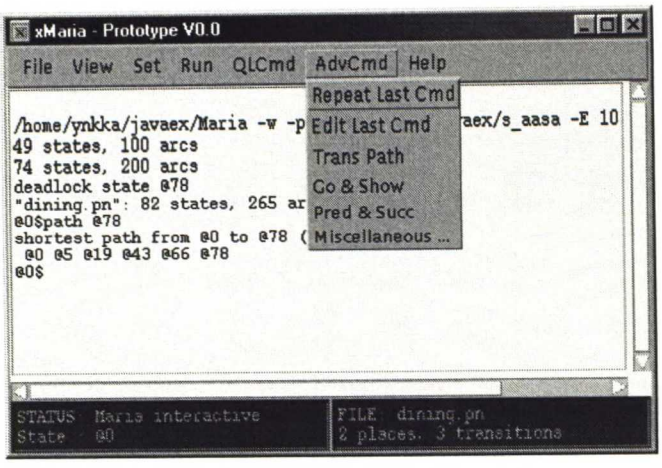


Kuva 29: MARIA:n komentojen valikko.

7.10. Erityiskomennot

Käyttöliittymään on ohjelmoitu joitakin erityiskomentoja, joihin päästään AdvCmd-valikosta (Advanced Commands).

Erityiskomentojen tarkoituksena on nopeuttaa tiettyjen usein toistuvien toimenpiteiden suoritusta saavutettavuusgraafin analysoinnissa. Tähän pyritään esimerkiksi kokoamalla erilaisia komentosarjoja erityiskomennoiksi, joiden suorituksen aikana ei myöskään yleensä tulosteta kaikkea MARIA:lta tulevaa tietoa näytölle.



Kuva 30: Erityiskomentojen valikko.

7.10.1. Edellisen komennon toisto

Viimeksi annettu komento toistetaan sellaisenaan valitsemalla Repeat Last Cmd.

7.10.2. Edellisen komennon toisto muokattuna

Viimeksi annettu komento toistetaan muokattuna valitsemalla Edit Last Cmd. Tällöin avataan syöttöikkuna (kappale 7.12), johon viimeisin komento otetaan automaattisesti muokattavaksi.

7.10.3. Transitiopolku

Valitsemalla TransPath saadaan näkyviin viimeksi talletettu polku transitioineen. Toiminto on kuvattu laajemmin kappaleessa 7.14.

7.10.4. Yhdistetyt komennot

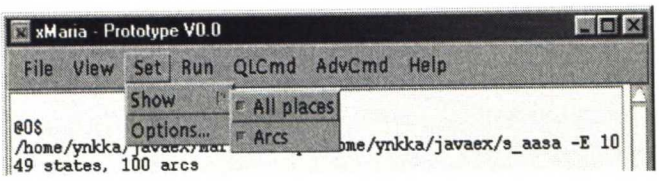
Komentoyhdistelmät Go & Show ja Pred & Succ on toteutettu ainoastaan esimerkkeinä mahdollisuudesta yhdistää Maria:n komentoja yhden valikkokomennon alle.

7.10.5. Komentojen kirjoittaminen

Minkä hyvänsä MARIA:n komennon voi kirjoittaa kokonaisuudessaan tekstimuodossa syöttöikkunaan (kappale 7.12), joka avautuu Miscellaneous-komennolla.

7.11. Asetukset

Valitsemalla Set ja Show saadaan näkyviin alivalikko, jolla ohjataan tietojen näyttöä eräiden MARIA:n komentojen tulostuksissa.

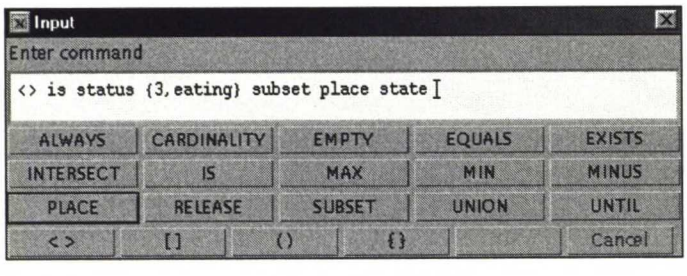


Kuva 31: Asetukset; tietojen näyttö.

Tässä alivalikossa kohteet ovat ns. päällä/pois-valintoja[1]. Jos ■ All places on valittuna, jokaisen paikan merkintä tulostetaan Show-komennolla, ts. yksittäisten paikkojen T/F-valinnalla (kappale 7.7) ei ole vaikutusta. Jos ■ Arcs on valittuna, transitioiden esikaarten muuttujien arvot tulostetaan Pred- ja Succ-komennoilla, muuten tulostetaan pelkät transitiot.

Options... avaa uuden ikkunan, jolla voidaan muuttaa käyttöliittymän oletusarvoja ja Maria:n komentoriviparametreja. Tätä toimintoa ei ole vielä toteutettu, koska työn aikana MARIA:n käyttö ei ollut siinä määrin vakiintunutta, että olisi voitu selkeästi yksilöidä parametroitavat ominaisuudet.

7.12. Syöttöikkuna



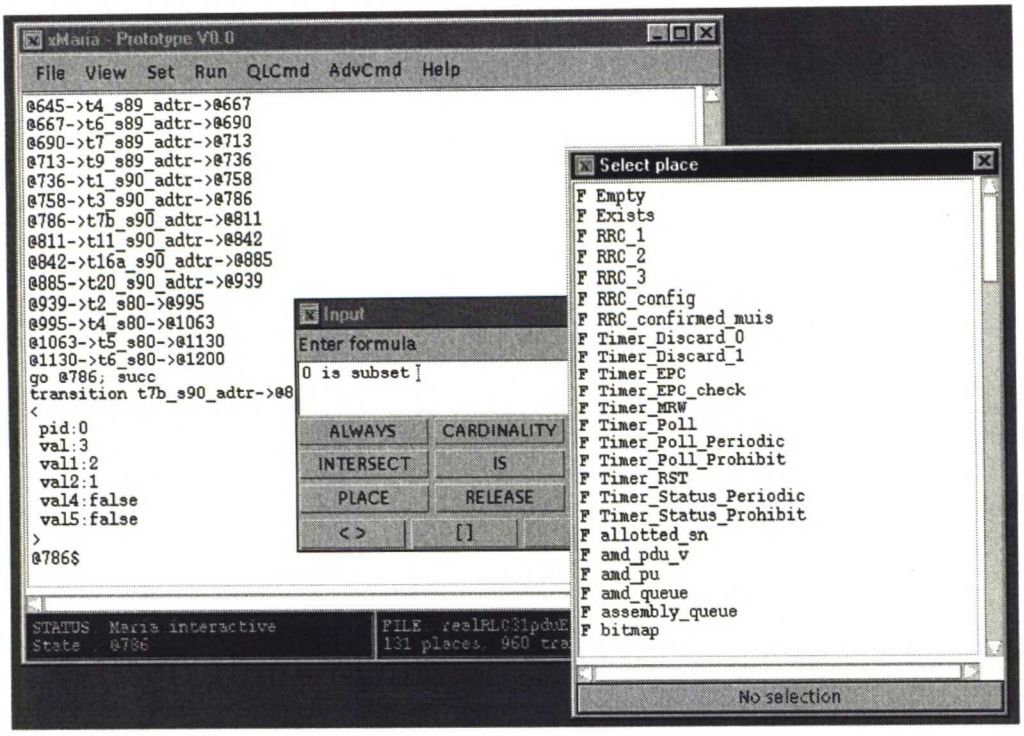
Kuva 32: Syöttöikkuna.

Joidenkin MARIA:n komentojen tarvitsemien parametrien syöttämiseksi avataan syöttöikkuna, jossa olevaan tekstikenttään käyttäjä kirjoittaa tarvittavat parametrit.

Syötön helpottamiseksi ja syntaksivirheiden välttämiseksi syöttöikkunaan on sisällytetty painikkeita, joilla tiettyjä avainsanoja ja pareittain tasapainotettuja sulkumerkkejä voidaan lisätä syötteeseen. Sulkuparit <>, [ ], ja ( ) toimivat samalla myös LTL-kaavojen (kappale 2.3.1) modaalioperaattoreina. Valittaessa painike PLACE avautuu seuraavassa kappaleessa kuvattu paikanvalintaikkuna.

[1] engl. checkbox. Ulkoasu vaihtelee ikkunointijärjestelmästä riippuen, eräs tavallinen versio on ☑.

7.13. Paikanvalintaikkuna



Kuva 33: Paikan nimen valinta paikanvalintaikkunasta.

Paikanvalintaikkuna poikkeaa edellä kappaleessa 7.7 kuvatusta paikkaluetteloikkunasta siten, että se avautuu vain syöttöikkunan PLACE-painikkeesta ja sulkeutuu heti, kun haluttu paikka on valittu tai, jos mitään paikan nimeä ei haluta syöttää, painettu No selection-painiketta.

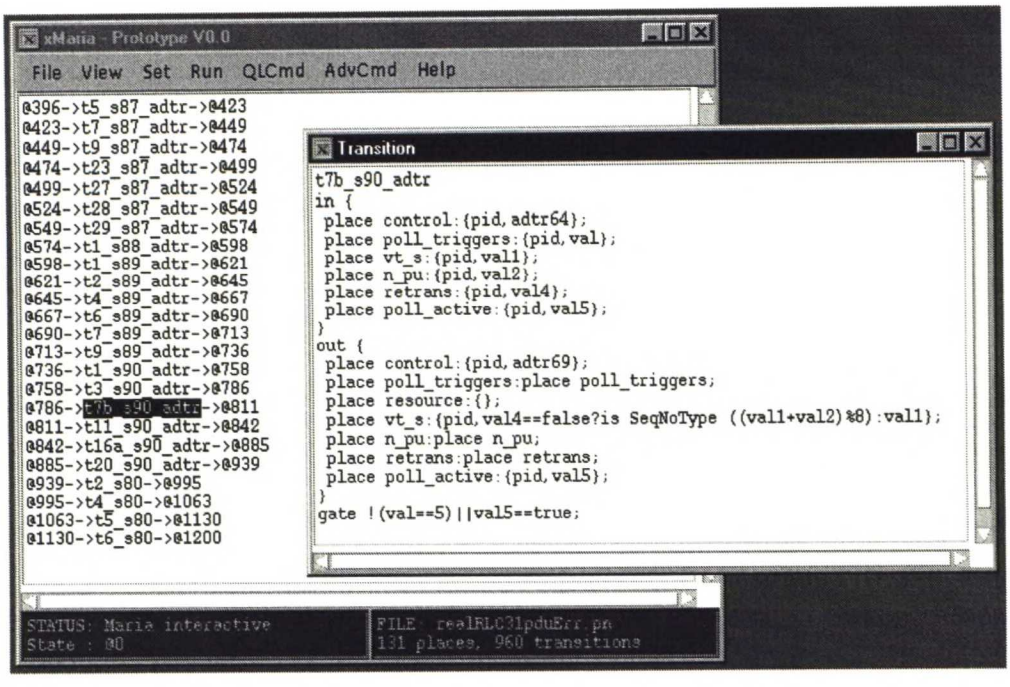
Kun paikanvalintaikkuna sulkeutuu, teksti `place` ja valittu paikan nimi lisätään automaattisesti syöttöikkunan tekstikenttään. Näin käyttäjän ei tarvitse muistaa paikkojen nimiä ja niiden oikeinkirjoitus on aina varmistettu.

7.14. Transitiopolku

Käytännön työssä käyttäjä on usein kiinnostunut erilaisista poluista saavutettavuusgraafin tilasta johonkin toiseen, esimerkiksi sellaiseen tilaan, jossa tapahtuu lukkiutuma, löytyy vastaesimerkki annetulle LTL-kaavalle tai joka muuten vain osoittautuu kiinnostavaksi. XMARIA tarjoaa polkujen tarkasteluun erityisen helppokäyttöisen työkalun, jota kutsutaan *transitiopoluksi*.

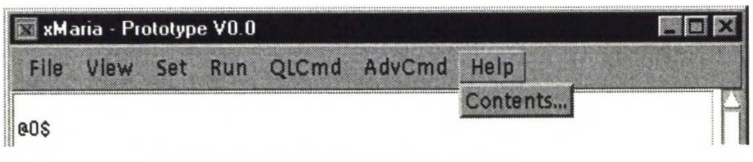
Ennen transitiopolun käyttöä XMARIA:lle on tehtävä tunnetuksi tarkasteltavaan polkuun kuuluvat tilat. Tämä voidaan tehdä Path-komennolla tai evaluoimalla jokin LTL-kaava, josta saadaan vastaesimerkkipolku. Viimeisin esiintynyt polku säilyy tallennettuna TransPath-komentoa varten, jolla saadaan loki-ikkunaan koko polku näkyville siten, että kutakin tilasiirtymää vastaava transitio tai joskus useampia samojen tilojen välillä esiintyviä transitioita myös näkyy.

Seuraavassa kuvassa näkyy loppuosa tilaan `@1200` johtaneesta polusta ja käyttäjä on halunnut tarkastella erityisesti siinä esiintynyttä transitiota **t7b_s90_adtr**. Samaan esimerkkiin liittyy myös edellä ollut Kuva 33, jossa on Go&Successors-komennolla tarkasteltu kaarimuuttujien arvoja.



Kuva 34: Transitiopolku ja valittu transitio MARIA-kielellä.

7.15. Käytön ohjeistus



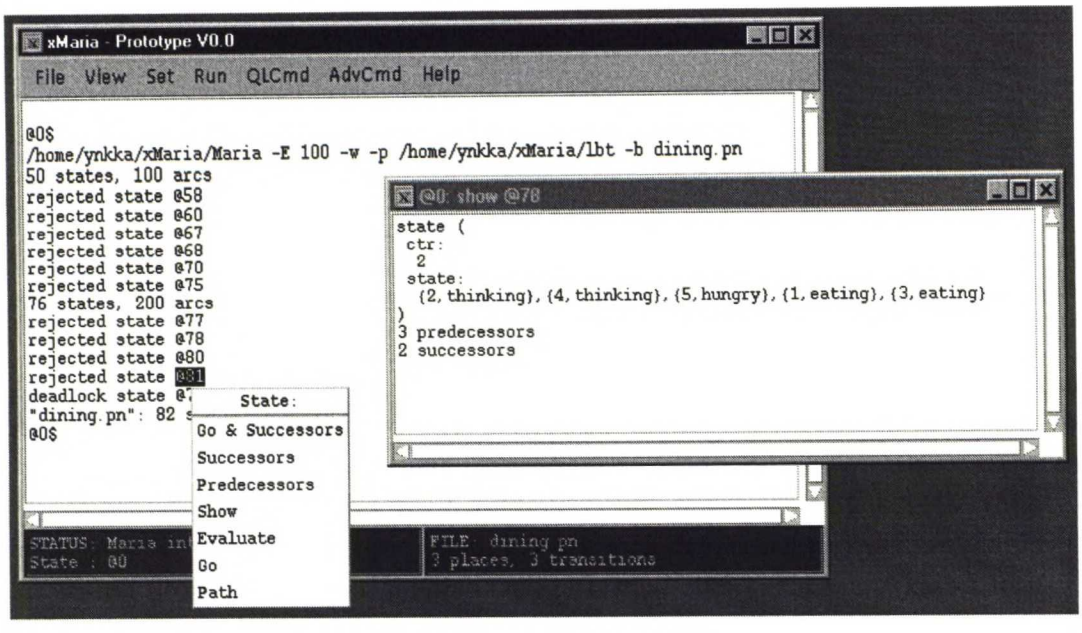
Kuva 35: Ohjeistuksen käynnistys.

XMARIA:n ohjeistus käynnistyy valitsemalla valikkoriviltä komento Help ja Contents. Ohjeistus on toteutettu Internet-sivuina, jotka esitetään UNIX-ympäristössä *Netscape Navigator*[1]-ohjelmalla. Ohjesivusto on esitetty liitteessä ja sitä voidaan selata myös itsenäisesti Internet-osoitteesta [XMH-W].

[1] Netscape Navigator on eräs yleisesti käytetty UNIX-järjestelmän selausohjelma. Mikä hyvänsä muukin vastaava selain voidaan valita käynnistettäväksi XMARIA:sta

7.16. Apuikkunan käyttöesimerkki

Kappaleessa 3.4 tarkasteltiin tilannetta, jossa viidestä filosofista kaksi syö yhtäaikaa ja todettiin, että yleensä helpointa on käydä läpi kaikkien kyseisen ehdon täyttävien tilojen merkinnät, jos halutaan tarkistaa, ettei kaksi vierekkäistä filosofia milloinkaan syö samanaikaisesti. XMARIA:n apuikkunan toiminta on suunniteltu siten, että se tarjoaa tämäntapaisiin ongelmiin ratkaisun.



Kuva 36: Apuikkunan käyttöesimerkki.

Osoittamalla kutakin lueteltua tilaa vuorollaan ja valitsemalla Show-komento ponnahdusvalikosta, tilan merkintä tulostuu apuikkunan entisen sisällön tilalle.

8. Yhteenveto

8.1. Tulosten tarkastelu

Tässä työssä on tutkittu nykyaikaisten käyttöliittymien soveltamista rinnakkaisten ja hajautettujen järjestelmien analysointiin formaalien menetelmien avulla. Käytännön kokemusten saamiseksi on toteutettu kokeellinen graafinen käyttöliittymä XMARIA, jonka avulla on käytetty TKK:n tietojenkäsittelyteorian laboratorion MARIA-analysaattoria.

Työn tekijä on itse käyttänyt XMARIA:a kehitystyön aikana jatkuvasti perehtyäkseen Petri-verkkomalleihin ja saavutettavuusanalyysiin, mm. tekemällä kurssin "Tik-79.179 *Rinnakkaiset ja hajautetut digitaaliset järjestelmät*" harjoitustyöt osittain sen avulla, mikä on suuresti vaikuttanut käyttöliittymän ominaisuuksiin. Kokemukset ovat tältä osin erittäin myönteiset, vaikkakin vertailukohta komentorivipohjaiseen käyttöliittymään on vähäinen.

Eräs laboratorion tutkija on käyttänyt XMARIA:a kehittäessään laajaa, käytännön tiedonsiirtoprotokollaa kuvaavaa Petri-verkkomallia. Tätä kirjoitettaessa malli sisälsi 131 paikkaa ja 960 transitiota ja sen saavutettavuusgraafi käsitti yli miljoona tilaa. Kyseisellä tutkijalla on pitkä kokemus sekä PROD:in että MARIA:n komentorivipohjaisesta käytöstä ja jo nyt saadut kokemukset graafisesta käyttöliittymästä ovat olleet erittäin myönteiset, mm. useita mallinnusvirheitä on paljastunut XMARIA:n transitiopolku-toiminnon avulla.

Lisäksi XMARIA on osoittautunut käyttökelpoiseksi muutettaessa EMMA:a kääntämään TNSDL-malleja MARIA:n kielelle. Muutostyötä tehtäessä mallin syntaksivirheet olivat erittäin tavallisia ja toisaalta EMMA:n generoimat mallit yksinkertaisimmillaankin kohtalaisen laajoja, joten XMARIA:n virheilmoituksia rajoittava toiminto osoittautui välttämättömäksi, sillä yleensä virheilmoituksia saatiin tuhansia.

EMMA:n muutostyö rajoitettiin tämän työn puitteissa minimiinsä, ts. toteutettiin ainoastaan PROD-mallien muunnos MARIA-kielelle, eikä MARIA:n tehokkaita tietotyyppejä otettu vielä käyttöön. Kokemukset antoivat kuitenkin viitteitä siitä, että EMMA:sta on kehitettävissä käyttökelpoinen TNSDL-esikäsittelijä. Tämä osa työstä osoitti myös välttämättömäksi MARIA:n käytettävyyden parantamisen, sillä EMMA-mallin sisältämien pitkien ja mutkikkaiden muuttujanimien käsittely komentorivipohjaisella käyttöliittymällä on käytännössä mahdotonta.

8.2. Jatkokehitysnäkymiä

Vaikka nyt aikaansaatu kokeellinen käyttöliittymä sisältää melkoisen määrän ominaisuuksia, on selvää, että paljon uusia toimintoja voisi vielä lisätä ja olemassa oleviakin muuttaa käyttökelpoisemmiksi. Ennen muutostyön aloittamista kannattaisi kuitenkin harkita jonkin muun kuin Javan AWT-kirjaston käyttöä ikkunoinnin toteutukseen, jolloin voitaisiin täysin hyödyntää nykyaikaisissa graafisissa käyttöliittymissä toteutettavissa olevia piirteitä. Esimerkiksi Javan uudempi Swing-kirjasto saattaisi olla parempi ratkaisu.

Yllättäväksi ongelmaksi XMARIA:n käyttöönotossa osoittautui eri järjestelmien Java-ajoympäristöjen erilaisuus, erityisesti se, etteivät jotkut ohjelman piirteet näytä lainkaan toimivan kaikissa kokeilluissa järjestelmissä. Jatkokehitystä silmälläpitäen on varminta ensin selvittää, mistä ongelmat johtuvat ja pyrkiä ratkaisemaan ne, muuten on etsittävä parempi toteutusvaihtoehto.

Alkuperäisenä ajatuksena oli liittää TNSDL-kääntäjän ohjaaminen osaksi käyttöliittymää, jolloin käyttäjä olisi halutessaan voinut käsitellä pelkästään TNSDL-kielistä lähdetiedostoa. Tällaista kääntäjää ei kuitenkaan saatu tämän työn aikana käyttöön, joten sen ja EMMA-kääntäjän liittäminen osaksi XMARIA:a on eräs potentiaalinen jatkokehityksen aihe. Itse EMMA-kääntäjän kehitys jatkuu välittömästi tämän työn jälkeen muiden henkilöiden toimesta mm. MARIA:n kehittyneiden tietotyyppien ja operaatioiden käyttöönotolla.

Analyysin tulosten esittäminen siten, että yhteys alkuperäiseen TNSDL-lähteeseen on helposti tunnistettavissa olisi eräs tärkeimmistä parannuskohteista. Tämä lienee suuritöinen, mutta kohtalaisen suoraviivainen tehtävä, sillä tarvittava selvitystyö on suurelta osin tehty aikoinaan EMMA:n tulostinohjelmaa toteutettaessa ja sen rutiinien muuttaminen Java-kielelle ei liene vaikeata.

Kirjallisuusviitteet

[EHS97] Jan Ellsberger, Dieter Hogrefe, Amardeo Sarma. *SDL - Formal Object-oriented Language for Communicating Systems*. Prentice Hall, London, 1997.

[Fla99] David Flanagan. *Java in a nutshell: a desktop quick reference*. 3rd ed. O'Reilly, Sebastopol CA, 1999.

[Gen87] Hartmann J. Genrich. Predicate/Transition Nets. In W. Brauer, W. Reisig and G. Rozenberg (eds), *Petri Nets: Central Models and Their Properties. Advances in Petri Nets 1986, part I.* LNCS vol 254. Springer-Verlag, Berlin, 1987.

[GTV93] Peter Grönberg, Mikko Tiusanen, Kimmo Varpaaniemi. *PROD – A Pr/T-Net Reachability analysis Tool*. Technical Report B11, Helsinki University of Technology, Department of Computer Science, Digital Systems Laboratory, June 1993.

[GJS96] James Gosling, Bill Joy, Guy Steele. *The Java Language Specification*. Addison-Wesley, 1996.

[HC97] Cay S. Horstmann, Gary Cornell. *Core Java 1.1. Volume 1: Fundamentals*. Sun Microsystems Press, Mountain View CA, 1997.

[HC98] Cay S. Horstmann, Gary Cornell. *Core Java 1.1. Volume 2: Advanced Features*. Sun Microsystems Press, Mountain View CA, 1998.

[HOPV95] Nisse Husberg, Leo Ojala, Olli-Matti Penttinen, Antti Vainonen. Report of a preliminary investigation: EMMA – an Extendible Multi Method Analyzer. Helsinki University of Technology, Department of Computer Science, Digital Systems Laboratory, August 1995. (Julkaisematon).

[Hus96] Nisse Husberg. SDL Modelling with High Level Petri Nets. In L. Czaja, P. Starke, H.D. Burkhard, and M. Lenz (eds.) *Workshop on Concurrency, Specification & Programming*, Berlin September 25-27 1996, vol 96 pp. 85-96. Humboldt Universität zu Berlin, Institut für Informatik, September 1996.

[HMJ96] Nisse Husberg, Markus Malmqvist, Tero Jyrinki. Emma: a Tool For Analysis of SDL Programs. Helsinki University of Technolo-

gy, Department of Computer Science, Digital Systems Laboratory, December 1996.

[Hus98] Nisse Husberg. The EMMA translator: From TNSDL to PROD. Helsinki University of Technology, Department of Computer Science, Digital Systems Laboratory, February 1998. (Julkaisematon).

[HM99] Nisse Husberg, Tapio Manner. Emma: Developing an Industrial Reachability Analyser for SDL. In J. M. Wing, J. Woodcock, J. Davies (eds.), *FM'99 – Formal Methods, World Congress on Formal Methods in the Development of Computing Systems*. Toulouse, France, September 20-24, 1999. LNCS 1708, pp. 642-661. Springer-Verlag, Berlin, Germany.

[ISO96] International Organization for Standardization. *Ergonomic requirements for office work with visual display terminals (VDTs) – Part 11: guidance on usability*. Standard ISO/DIS 9241-11-2. 1996.

[ITU88] International Telecommunication Union (publ.). *Specification and Description Language (SDL)*. ITU-T Recommendation Z.100. ITU, Geneve, 1988.

[ITU99] International Telecommunication Union (publ.). *Specification and Description Language (SDL)*. ITU-T Recommendation Z.100. ITU, Geneve, 1999.

[Jen97] Kurt Jensen. *Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use*. Volume 1. 2nd ed. Springer-Verlag, Berlin, 1997.

[Jyr97] Tero Jyrinki. *Dynamical analysis of SDL programs with Predicate/Transition nets*. Technical Report B17, Helsinki University of Technology, Department of Computer Science, Digital Systems Laboratory, April 1997.

[Kal95] Anna Kalimo (toim.). *Graafisen käyttöliittymän suunnittelu. Opas ohjelmistojen käytettävyyteen*. Tietotekniikan kehittämiskeskus Tieke ry. (julk.). Suomen ATK-kustannus, Espoo 1995.

[Kal92] Titti Kallio. *Käyttöliittymät ja niiden suunnittelu*. Suomen ATK-kustannus Oy, Espoo 1992.

[KV98] Ekkart Kindler, Hagen Völzer. Flexibility in algebraic nets. In Jörg Desel and Manuel Silva (eds), *Application and Theory of Petri*

Nets 1998, 19th International Conference, ICATPN'98, Lisbon, Portugal, June 1998. LNCS vol 1420, pp. 345-364. Springer-Verlag, Berlin, Germany.

[Koi01] Jani Koivulainen. Upgrading EMMA to support TNSDL 4.0. Tik-79.189 Tietojenkäsittelyteorian erikoistyö. Teknillinen Korkeakoulu, tietojenkäsittelyteorian laboratorio, toukokuu 2001.

[LHK01] Johan Lilius, Keijo Heljanko, Esa Kettunen. Tik-79.179 Rinnakkaiset ja hajautetut digitaaliset järjestelmät. Luentomonisteet. Teknillinen Korkeakoulu, tietojenkäsittelyteorian laboratorio, 2001.

[LRKT95] Markus Lindqvist, Erkki Ruohtula, Esa Kettunen, Heikki Tuominen. *The TNSDL Book*. Standard Operating Procedure YFR 0811/3E. Nokia Telecommunications, November 1995.

[Mal97] Markus Malmqvist. *Methodology of Dynamical Analysis of SDL Programs with Predicate/Transition Nets*. Technical Report B16, Helsinki University of Technology, Department of Computer Science, Digital Systems Laboratory, April 1997.

[Mic99] Microsoft Corporation (anon.). *Visual Basic 6 – ohjelmoijan käsikirja*. IT Press, Helsinki, 1999.

[Mäk98] Marko Mäkelä. *Implementing the Front-End of an SDL Compiler*. Master's Thesis. Helsinki University of Technology, Department of Computer Science and Engineering, Laboratory for Theoretical Computer Science, November 1998.

[Mäk00] Marko Mäkelä. *A Reachability Analyser for Algebraic System Nets*. Licentiate's Thesis. Helsinki University of Technology, Department of Computer Science and Engineering, Laboratory for Theoretical Computer Science, March 2000.

[Mäk01] Marko Mäkelä. *Maria: Modular reachability analyzer for algebraic system nets*. Version 1.0 documentation. Helsinki University of Technology, Department of Computer Science and Engineering, Laboratory for Theoretical Computer Science, November 2001.

[PM00] Juha Peltomäki, Pekka Malmirae. *JAVA. Java-ohjelmoinnin peruskirja*. 2. painos. Teknolit Oy, Jyväskylä, 2000.

[Pel01] Doron A. Peled. *Software reliability methods*. Springer-Verlag, New York, 2001.

[Pre94] Jenny Preece et al. *Human-Computer Interaction*. Addison-Wesley Publishing Company, Wokingham, 1994.

[Rei91] Wolfgang Reisig. Petri nets and algebraic specifications. In Theorethical Computer Science, 80: 1-34, May 1991.

[URG00] Anon. *Tik-86.126 GUI Standard and Guidelines*. Helsinki University of Technology, Usability Research Group, February 2000. (Julkaisematon).

[VHHP95] Kimmo Varpaaniemi, Jaakko Halme, Kari Hiekkanen, Tino Pyssysalo. *PROD Reference Manual*. Technical Report B13, Helsinki University of Technology, Department of Computer Science, Digital Systems Laboratory, August 1995.

[Wik99] Arto Wikla. *Ohjelmoinnin perusteet Java-kielellä*. 2. painos. Ota-DATA ry, Espoo, 1999.

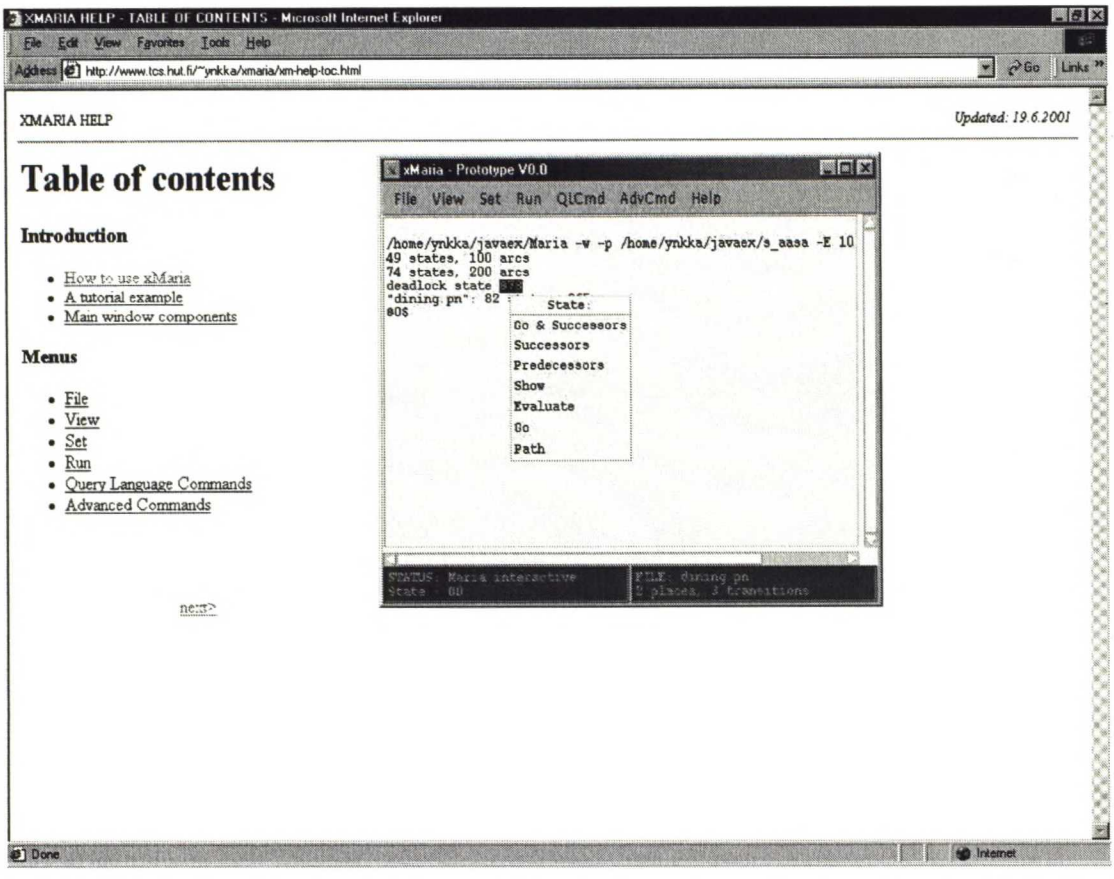
Internet-viitteet

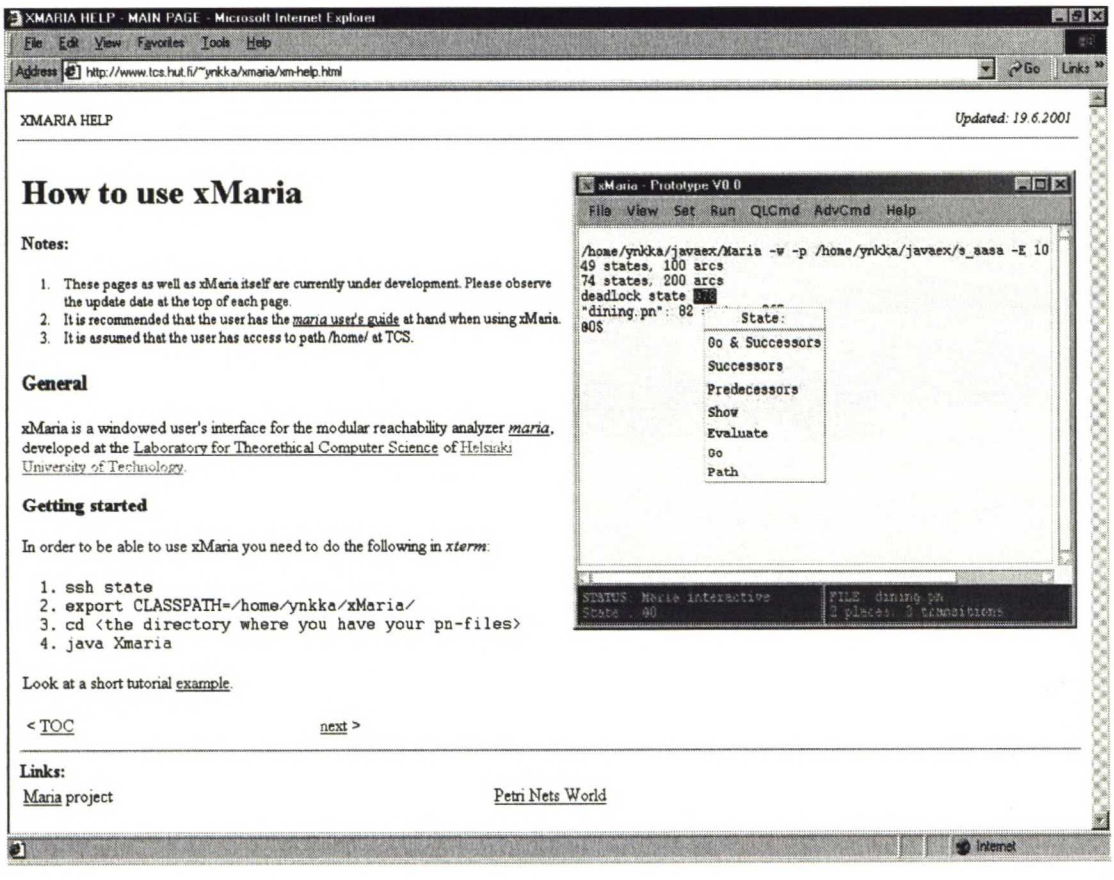
[API-W] http://java.sun.com/products/jdk/1.2/docs/api/

[MAR-W] http://www.tcs.hut.fi/maria/

[XMH-W] http://www.tcs.hut.fi/~ynkka/xmaria/xm-help.html

Liite A: XMARIA:n ohjesivut





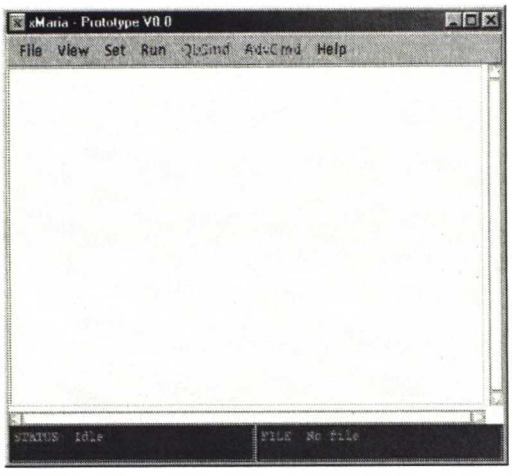
XMARIA HELP - MAIN WINDOW COMPONENTS - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Address http://www.tcs.hut.fi/~ynkka/xmaria/xm-help-mwin.html

XMARIA HELP *Updated: 19.6.2001*

Main window components



Note: All xMaria windows are resizable and movable in order to let the user make best use of the screen.

Menu bar

contains menus to enter commands to xMaria as well as to *maria*.

Log area

is used to display a log of the analysis process.

Status display

displays current status of *maria*.

File display

displays the name and statistics of the current source file.

< prev
< TOC

next >

Done Internet

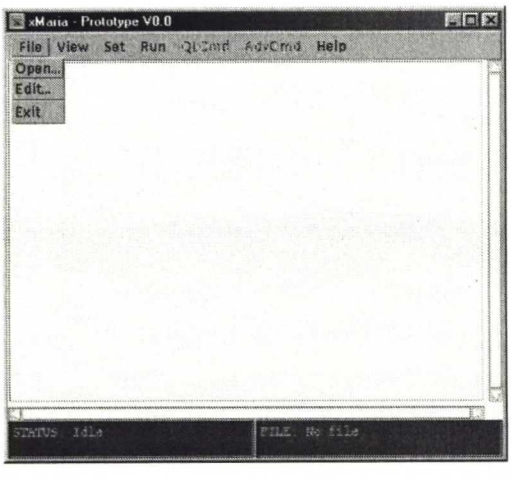
XMARIA HELP - FILE MENU - Microsoft Internet Explorer

File Edit View Favorites Tools Help

Address http://www.tcs.hut.fi/~ynkka/xmaria/xm-help-file.html

XMARIA HELP *Updated: 19.6.2001*

File menu

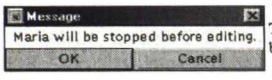


Open

Opens a dialog window listing all pn-files in the current directory. After the file has been selected its syntax is checked by *maria* and internal symbol tables are updated.

Edit

A text editor (currently *Emacs*) is started with the current source file.



The user is first notified that *maria* has to be stopped and restarted.

After editing the file is automatically reopened.

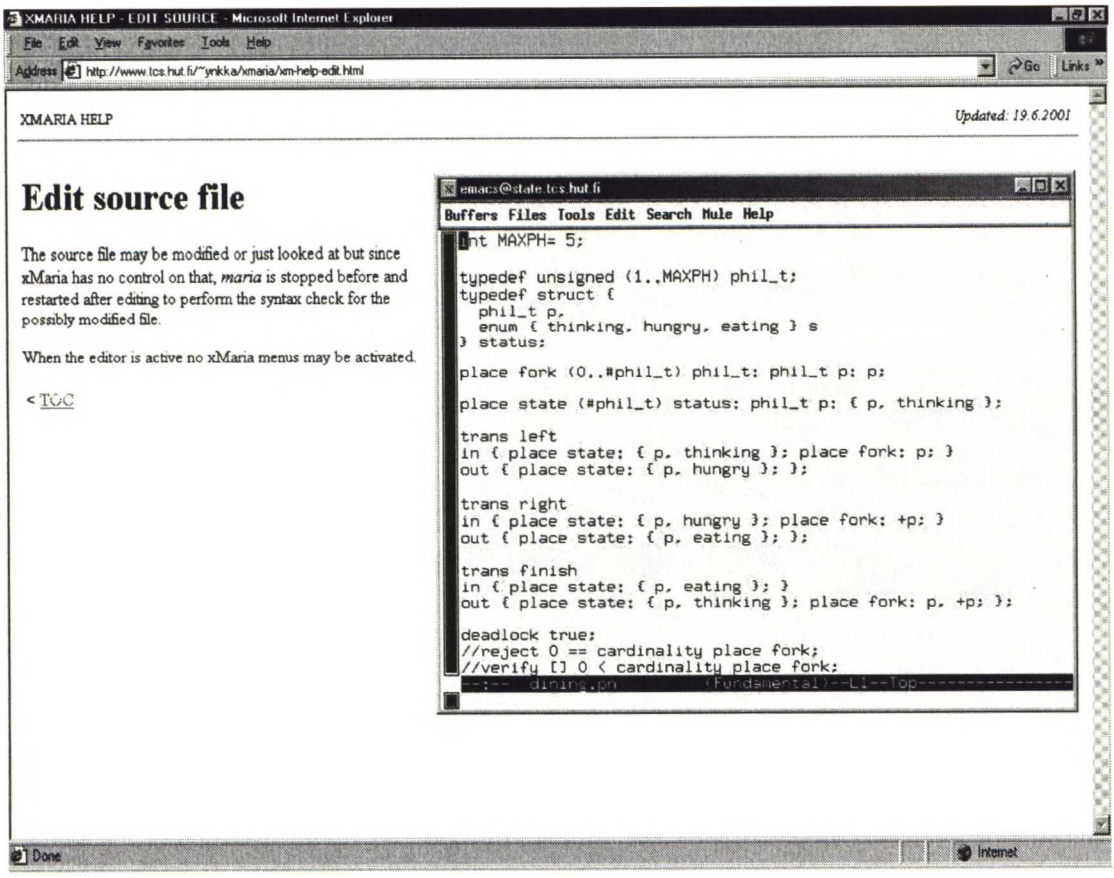
Exit

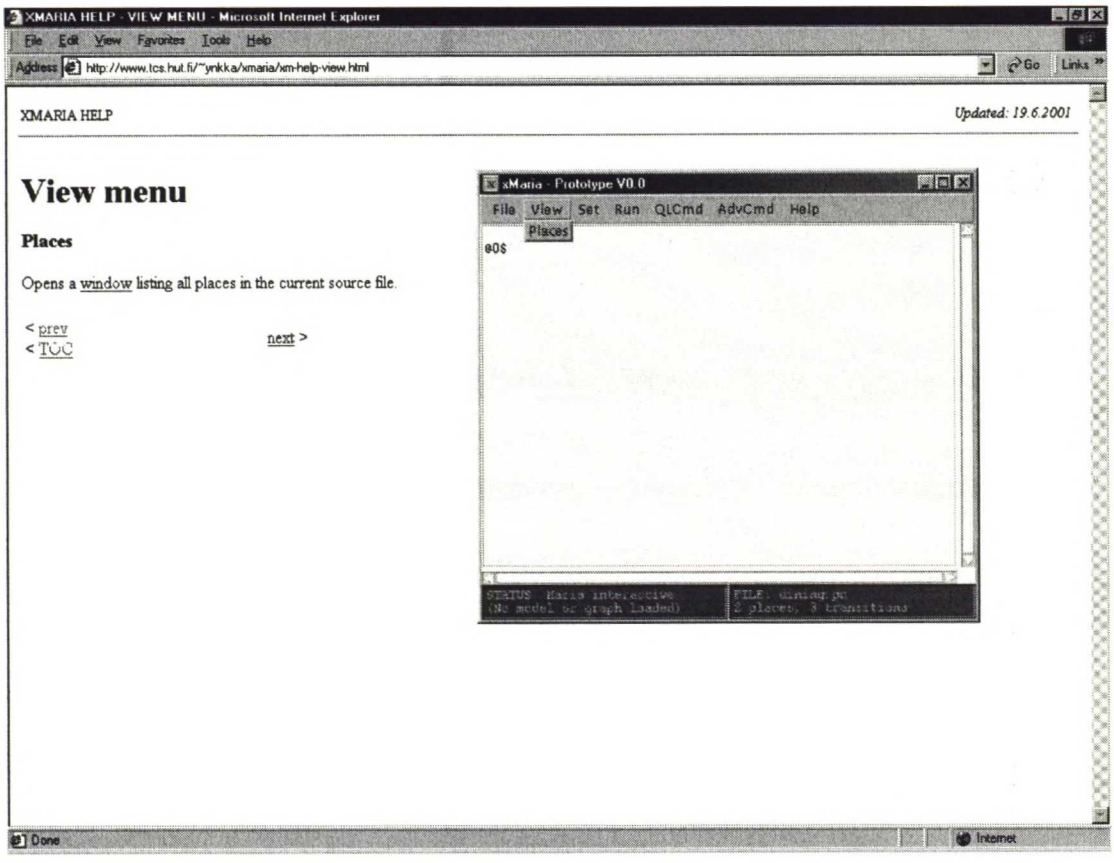
Stops xMaria (and *maria* if active).

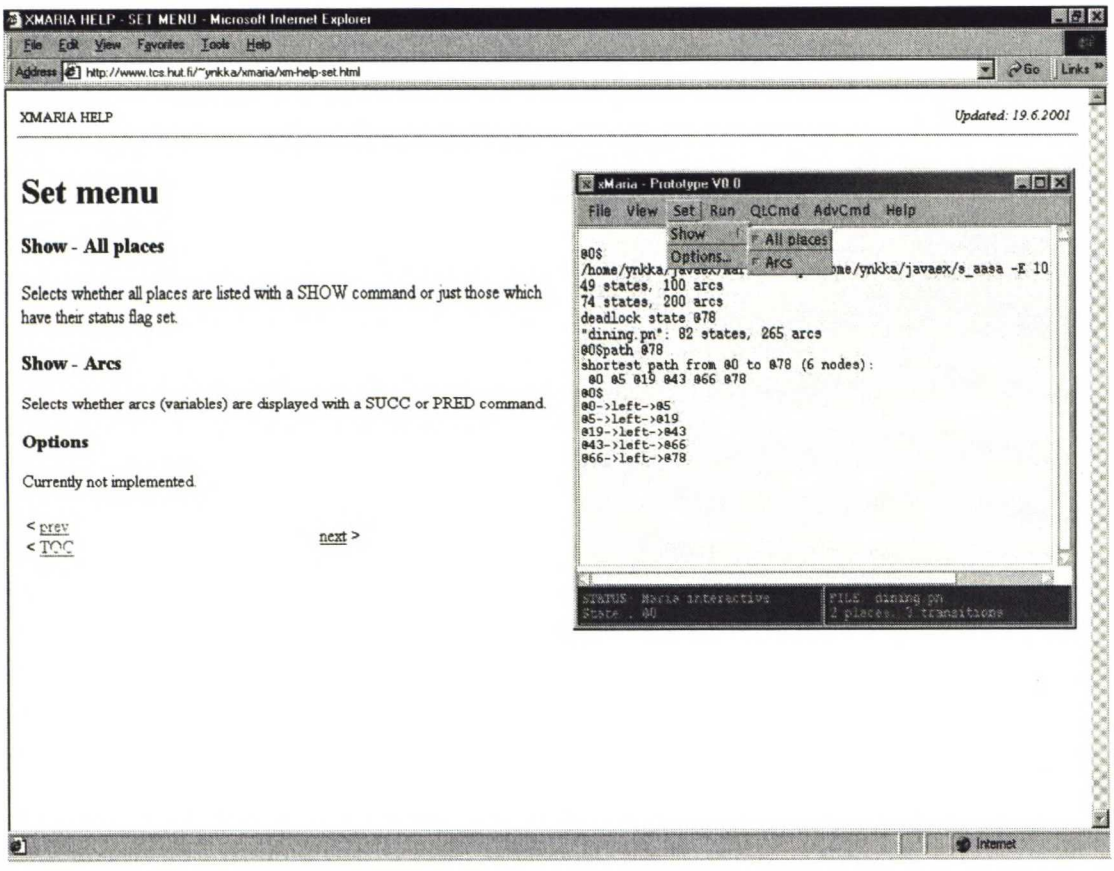
< prev
< TOC

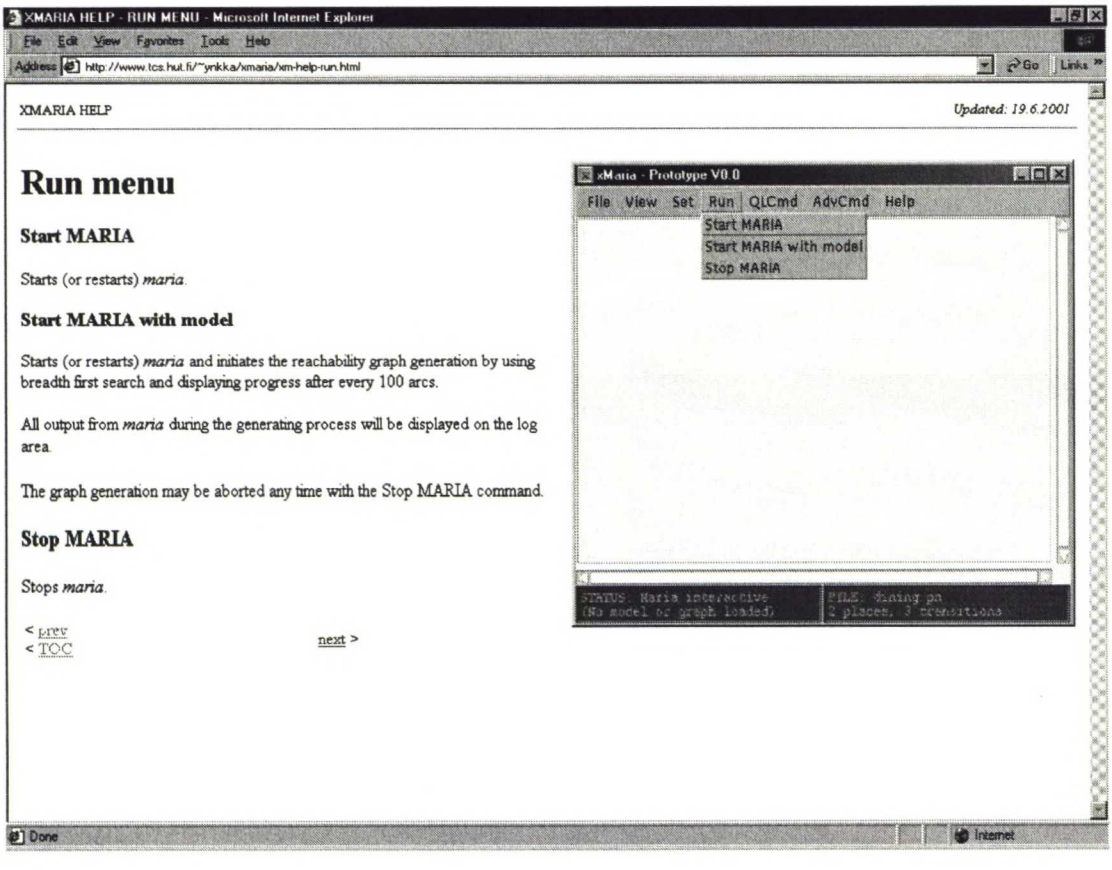
next >

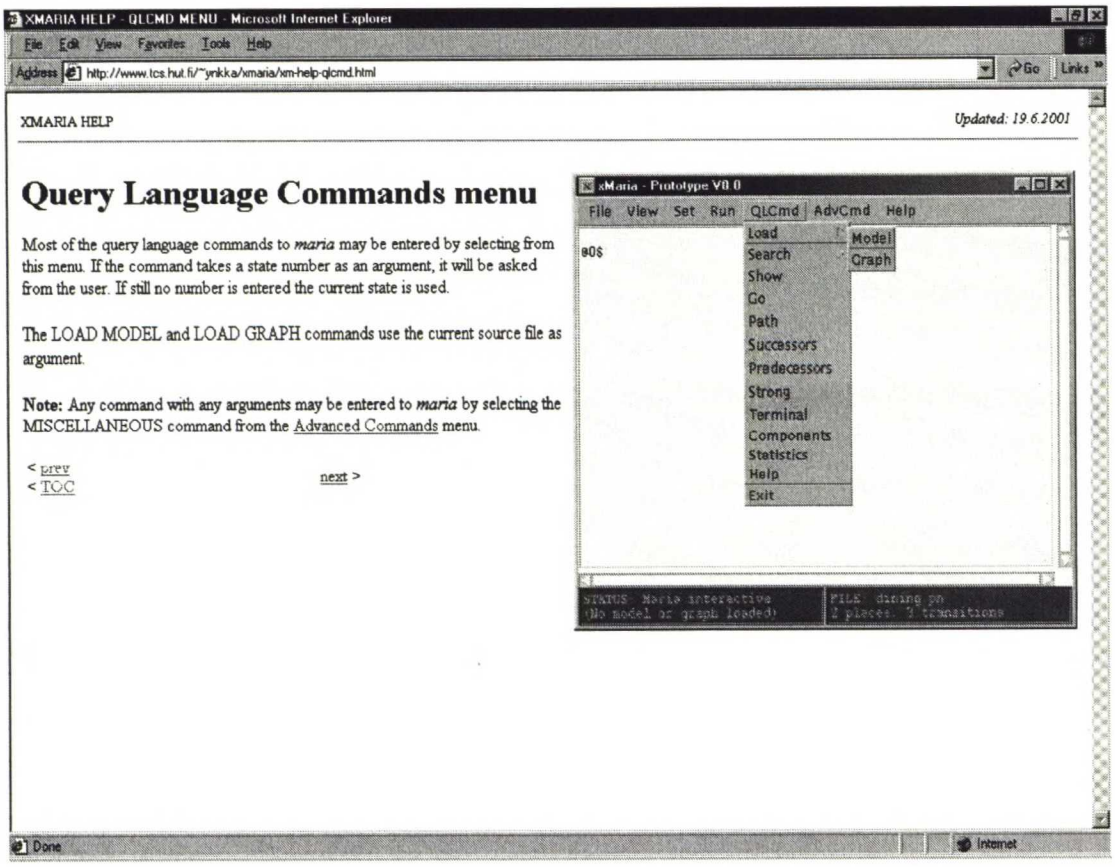
Done Internet

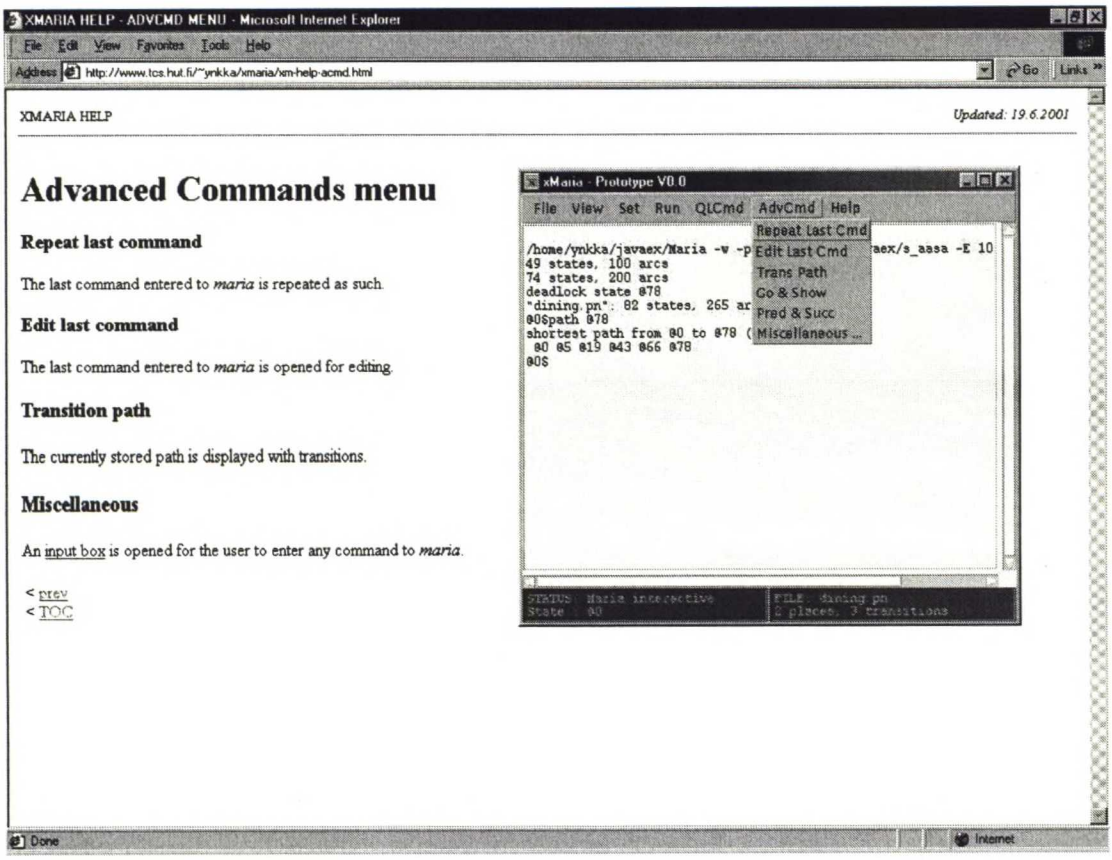

XMARIA HELP - EDIT SOURCE - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://www.tcs.hut.fi/~ynkka/xmaria/xm-help-edit.html
Go Links
XMARIA HELP
Updated: 19.6.2001
Edit source file
The source file may be modified or just looked at but since xMaria has no control on that, maria is stopped before and restarted after editing to perform the syntax check for the possibly modified file.
When the editor is active no xMaria menus may be activated.
< TOC
emacs@state.tcs.hut.fi
Buffers Files Tools Edit Search Mule Help
nt MAXPH= 5;
typedef unsigned (1..MAXPH) phil_t;
typedef struct {
phil_t p,
enum { thinking, hungry, eating } s
} status;
place fork (0..#phil_t) phil_t: phil_t p: p;
place state (#phil_t) status: phil_t p: { p, thinking };
trans left
in { place state: { p, thinking }; place fork: p; }
out { place state: { p, hungry }; };
trans right
in { place state: { p, hungry }; place fork: +p; }
out { place state: { p, eating }; };
trans finish
in { place state: { p, eating }; }
out { place state: { p, thinking }; place fork: p, +p; };
deadlock true;
//reject 0 == cardinality place fork;
//verify [] 0 < cardinality place fork;
--:--- dining.pn (Fundamental)--L1--Top
Done
Internet


XMARIA HELP - VIEW MENU - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://www.tcs.hut.fi/~ynkka/xmaria/xm-help-view.html
Go Links
XMARIA HELP
Updated: 19.6.2001
View menu
Places
Opens a window listing all places in the current source file.
< prev
< TOC
next >
xMaria - Prototype V0.0
File View Set Run QLCmd AdvCmd Help
Places
@0$
STATUS Maria interactive
(No model or graph loaded)
FILE: dining.pn
5 places, 5 transitions
Done
Internet


XMARIA HELP - SET MENU - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://www.tcs.hut.fi/~ynkka/xmaria/xm-help-set.html
Go Links
XMARIA HELP
Updated: 19.6.2001
Set menu
Show - All places
Selects whether all places are listed with a SHOW command or just those which have their status flag set.
Show - Arcs
Selects whether arcs (variables) are displayed with a SUCC or PRED command.
Options
Currently not implemented.
< prev
< TOC
next >
xMaria - Prototype V0.0
File View Set Run QtCmd AdvCmd Help
Show
Options...
All places
Arcs
@0$
49 states, 100 arcs
74 states, 200 arcs
deadlock state @78
"dining.pn": 82 states, 265 arcs
@0$path @78
shortest path from @0 to @78 (6 nodes):
@0 @5 @19 @43 @66 @78
@0$
@0->left->@5
@5->left->@19
@19->left->@43
@43->left->@66
@66->left->@78
STATUS: Maria interactive
State : @0
FILE: dining.pn
2 places, 3 transitions
Internet


XMARIA HELP - RUN MENU - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://www.tcs.hut.fi/~ynkka/xmaria/xm-help-run.html
Go Links
XMARIA HELP
Updated: 19.6.2001
Run menu
Start MARIA
Starts (or restarts) maria.
Start MARIA with model
Starts (or restarts) maria and initiates the reachability graph generation by using breadth first search and displaying progress after every 100 arcs.
All output from maria during the generating process will be displayed on the log area.
The graph generation may be aborted any time with the Stop MARIA command.
Stop MARIA
Stops maria.
< prev
< TOC
next >
xMaria - Prototype V0.0
File View Set Run QtCmd AdvCmd Help
Start MARIA
Start MARIA with model
Stop MARIA
STATUS: Maria interactive
(No model or graph loaded)
FILE: dining.pn
2 places, 3 transitions
Done
Internet


XMARIA HELP - QLCMD MENU - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://www.tcs.hut.fi/~ynkka/xmaria/xm-help-qlcmd.html
Go Links
XMARIA HELP
Updated: 19.6.2001
Query Language Commands menu
Most of the query language commands to maria may be entered by selecting from this menu. If the command takes a state number as an argument, it will be asked from the user. If still no number is entered the current state is used.
The LOAD MODEL and LOAD GRAPH commands use the current source file as argument.
Note: Any command with any arguments may be entered to maria by selecting the MISCELLANEOUS command from the Advanced Commands menu.
< prev
< TOC
next >
xMaria - Prototype V0.0
File View Set Run QLCmd AdvCmd Help
@0$
Load
Search
Show
Go
Path
Successors
Predecessors
Strong
Terminal
Components
Statistics
Help
Exit
Model
Graph
STATUS: Maria interactive
(No model or graph loaded)
FILE: dining.pn
2 places, 3 transitions
Done
Internet


XMARIA HELP - ADVCMD MENU - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Address http://www.tcs.hut.fi/~ynkka/xmaria/xm-help-acmd.html
Go Links
XMARIA HELP
Updated: 19.6.2001
Advanced Commands menu
Repeat last command
The last command entered to maria is repeated as such.
Edit last command
The last command entered to maria is opened for editing.
Transition path
The currently stored path is displayed with transitions.
Miscellaneous
An input box is opened for the user to enter any command to maria.
< prev
< TOC
xMaria - Prototype V0.0
File View Set Run QLCmd AdvCmd Help
Repeat Last Cmd
Edit Last Cmd
Trans Path
Go & Show
Pred & Succ
Miscellaneous ...
/home/ynkka/javaex/Maria -w -p
aex/s_aasa -E 10
49 states, 100 arcs
74 states, 200 arcs
deadlock state @78
"dining.pn": 82 states, 265 ar
@0$path @78
shortest path from @0 to @78 (
@0 @5 @19 @43 @66 @78
@0$
STATUS: Maria interactive
State : @0
FILE: dining.pn
2 places, 3 transitions
Done
Internet